TURING 图灵程序设计丛书

Packt

Deep Learning for Computer Vision
Expert techniques to train advanced neural networks using TensorFlow and Keras

计算机视觉之深度学习

使用TensorFlow和Keras训练高级神经网络

[英] 拉贾林加帕·尚穆加马尼◎著　白勇◎译

快速入门计算机视觉应用开发，涉及多种深度学习算法及其实现

以真实数据集为例，结合流行技术框架

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

作者简介

拉贾林加帕·尚穆加马尼
(Rajalingappaa Shanmugamani)

目前在Kairos担任技术经理。在此之前，曾在新加坡SAP公司担任深度学习主管，也在创业公司从事过计算机视觉产品的开发和咨询工作。在同行评审的期刊和会议上多次发表文章，并在机器学习领域申请了专利。与他人合著出版了*Hands-On Natural Language Processing with Python*、*Python Reinforcement Learning*、*Python: Advanced Guide to Artificial Intelligence*、*TensorFlow Deep Learning Projects*等书。

译者简介

白勇

海南大学信息科学技术学院教授、博士生导师、美国归国博士。长期从事物联网、人工智能方面的研究。已承担国家级和省部级项目10多项，发表学术论文100多篇，授权中国和美国发明专利10多项。

数字版权声明

TURING 图灵程序设计丛书

Deep Learning for Computer Vision

Expert techniques to train advanced neural networks using TensorFlow and Keras

计算机视觉之深度学习

使用TensorFlow和Keras训练高级神经网络

[英] 拉贾林加帕·尚穆加马尼◎著　　白勇◎译

人民邮电出版社

北　京

图书在版编目（CIP）数据

计算机视觉之深度学习 : 使用TensorFlow和Keras训练高级神经网络 / (英) 拉贾林加帕·尚穆加马尼著 ; 白勇译. -- 北京 : 人民邮电出版社, 2020.1
（图灵程序设计丛书）
ISBN 978-7-115-53158-2

Ⅰ. ①计… Ⅱ. ①拉… ②白… Ⅲ. ①计算机视觉②机器学习 Ⅳ. ①TP302.7②TP181

中国版本图书馆CIP数据核字(2019)第301887号

内 容 提 要

本书开门见山，直接帮助你准备好训练高效深度学习模型的环境，以完成各种计算机视觉任务。书中介绍了常见的深度学习架构，如卷积神经网络和递归神经网络；讲述了如何利用深度学习进行图像分类、图像检索、目标检测、语义分割等内容。读完本书，你将能够开发和训练自己的深度学习模型，并用它们解决计算机视觉难题。

本书适合所有对计算机视觉和深度学习感兴趣的读者。

◆ 著　　　[英] 拉贾林加帕·尚穆加马尼
译　　　白　勇
责任编辑　张海艳
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　印刷

◆ 开本：800×1000　1/16
印张：13.5
字数：319千字　　2020年1月第1版
印数：1－3 500册　　2020年1月北京第1次印刷

著作权合同登记号　图字：01-2018-2241号

定价：59.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

译 者 序

计算机视觉是一门研究如何使机器“看”的科学。它涉及如何使计算机从数字图像或视频中获得高层次的理解，并寻求自动化人类视觉系统可以完成的任务。深度学习是机器学习研究中的一个新领域，它通过多层神经网络模仿人脑的机制来分析和解释数据。深度学习不仅推动了计算机视觉领域的发展，其自身也在解决计算机视觉问题的过程中不断进步。结合深度学习，计算机视觉的许多新应用应运而生，并且正在成为我们日常生活的一部分，包括人脸识别、自动驾驶汽车中的机器视觉等。

本书系统概述了基于深度学习的计算机视觉的相关理论和技术现状，涵盖了计算机视觉深度学习的主要问题，包括图像分类、图像检索、目标检测、语义分割、相似性学习、图像题注、生成模型、视频分类、平台部署。本书针对每个问题介绍了常见的应用场景、数据集、主要技术和 TensorFlow 框架下的编程实例。通过阅读本书和代码实践，读者可以比较全面地了解基于深度学习的计算机视觉的基础知识和领先技术。

本书的译文力求忠实准确地表达原文。相关术语的翻译参考了业内的一般用语。由于计算机视觉深度学习领域的发展日新月异，新技术和新术语也层出不穷。对新术语或读者可能较陌生的术语，译文中给出了相应的英文术语对照。另外，书中诸如“当前最好的技术”等说法，会随着新技术的出现而发生变化。读者应持续关注最新的研究进展。

本书翻译过程中，图灵公司的杨琳编辑审阅了全部书稿，提出了许多宝贵意见，对本书质量的提高有很大帮助，在此表示衷心的感谢。

由于译者水平所限，书中难免有错误和不当之处，欢迎专家和读者批评指正。邮件请发至 baiyong666ai@qq.com。另外，读者可加入本书的 QQ 交流群（群号：891515375）。

序

深度学习正在使人工智能发生革命性的变化。在未来的几十年里，它将彻底改变世界。在深度学习的推动下，人工智能将有望与工业革命相提并论。这让人们对未来充满兴奋，同时也感到恐惧。但实际上，就像工业革命和机械一样，深度学习将提高工业生产能力并显著改善人类生活水平。它不会取代人们的工作，而会创造更多更高水平的工作机会。这就是本书如此重要和及时的原因。本书将介绍基于深度学习的计算机视觉，包括它的功能和众多应用，从而为读者面对未来十年里将呈指数级增长的新兴产业打下基础。

拉贾林加帕·尚穆加马尼是一位出色的研究人员。我曾与他在几个计算机视觉项目中合作过。他作为首席工程师设计并交付了一个用于时尚搜索的复杂计算机视觉和深度学习系统，该系统的实际部署取得了巨大成功。他的强项之一是，能够针对复杂问题进行最先进的研究并将其应用于现实世界。正如本书展示的那样，他还可以分解复杂的想法并用简单的术语来解释。拉贾抱负远大，拥有职业道德。在本书中，他对使用深度学习的计算机视觉现状进行了很好的概述，这是许多人望尘莫及的。本书是拉贾的一项伟大成就，我相信读者在未来几年中都会从本书中获得阅读享受并受益匪浅。

Stephen Moore 博士

EmotionReader 首席技术官，于新加坡

前　　言

本书面向想要掌握基于深度学习的计算机视觉技术的读者，帮助他们实现诸多应用。本书将为读者研发基于计算机视觉的产品提供相应的工具和技术。为此，全书采取理论讲解和实例介绍相结合的写作方式。

读者对象

本书适合有意探究如何将深度学习应用于计算机视觉问题（如分类、检测、检索、分割、生成、题注和视频分类）的读者，以及希望了解如何在各种限制条件（如较少的数据、不平衡的类别、噪声）下实现较高准确性的读者。本书还适合想知道如何在各种平台（AWS、Google Cloud、Raspberry Pi 和手机）上部署经过训练的模型的读者。通过学习本书，读者应该能够针对人物检测、人脸识别、产品搜索、医学图像分割、图像生成、图像题注、视频分类等问题开发相应的代码。

本书内容

第 1 章“**入门**”，介绍深度学习的基础知识以及相关术语。这一章将指导读者安装必要的软件包，为学习后续章节做准备。

第 2 章“**图像分类**”，讨论图像分类问题，即如何将图像作为一个整体打标签。我们将学习图像分类技术，并训练一个用于宠物分类的深度学习模型。此外还会学习提高图像分类准确性的方法，并深入探究各种先进的图像分类架构。

第 3 章“**图像检索**”，介绍图像深层特征和图像检索。我们将学习获取模型可视化、视觉特征以及使用 TensorFlow Serving 进行推断的各种方法，并且使用视觉特征进行产品检索。

第 4 章“**目标检测**”，讨论如何检测图像中的目标对象。我们将学习目标检测的各种技术，并将其应用于行人检测。这一章将使用与物体检测相关的 TensorFlow API。

第 5 章“**语义分割**”，介绍像素级图像分割。我们将学习图像分割技术，并训练一个医学图像分割模型。

第 6 章“**相似性学习**”，讲述相似性学习。我们将学习相似性匹配以及如何训练人脸识别模型。这一章将展示一个训练人脸关键点的模型。

第 7 章“**图像题注**”，介绍如何为图像生成或选择题注。我们将学习自然语言处理技术以及如何使用这些技术为图像生成题注。

第 8 章“**生成模型**”，讨论如何为各种目的生成合成图像。我们将学习什么是生成模型并将其用于图像生成应用程序，如风格迁移、训练数据等。

第 9 章“**视频分类**”，介绍面向视频数据的计算机视觉分类技术。我们将了解解决视频问题与图像问题的主要区别以及如何实现视频分类技术。

第 10 章“**部署**”，讨论深度学习模型的部署步骤。我们将学习如何部署训练好的模型并针对各种平台优化执行速度。

如何充分利用本书

本书中的例子可以在 Windows、Ubuntu 或 Mac 上运行，所有安装说明都包含在内。Python 和机器学习的基础知识是必需的。读者最好有 GPU 硬件，但这不是必需的。

下载示例代码文件

你可以从 www.packtpub.com 上的账户下载本书的示例代码文件。如果你在其他地方购买了本书，可以访问 www.packtpub.com/support 并注册，文件会直接通过电子邮件发送给你。

你可以按照以下步骤下载代码文件：

(1) 登录或注册www.packtpub.com；
(2) 选择 SUPPORT 选项卡；
(3) 点击 Code Downloads & Errata；
(4) 在 Search 框中输入书名，然后按照屏幕上的说明进行操作。

下载文件后，确保使用最新版本的软件来解压缩：

- 适用于 Windows 的 WinRAR/7-Zip
- 适用于 Mac 的 Zipeg/iZip/UnRarX
- 适用于 Linux 的 7-Zip/PeaZip

本书的代码包也托管在 GitHub 上：https://github.com/PacktPublishing/Deep-Learning-for-Computer-Vision。我们在 https://github.com/PacktPublishing/上还提供了许多其他图书和视频的代码包。去看一下吧！

排版约定

本书使用了不同的文本样式来区分不同种类的信息。

正文中的代码采用以下样式："请注意，图形是使用 `summary_writer` 写入的。"

代码块的样式如下：

```
merged_summary_operation = tf.summary.merge_all()
train_summary_writer = tf.summary.FileWriter('/tmp/train', session.graph)
test_summary_writer = tf.summary.FileWriter('/tmp/test')
```

命令行输入或输出的样式如下：

```
wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
```

黑体字：表示本书中出现的新术语或重要的词语。屏幕菜单或对话框中的文字按照如下样式显示："完成后，点击 Actions|Instance State|Terminate 终止实例。"

警告或重要说明如此图标所示。

提示和技巧如此图标所示。

保持联系

我们欢迎读者反馈。

一般反馈：发送电子邮件到 feedback@packtpub.com 并在消息主题中提及书名。如果你对本书的任何方面有疑问，请发送电子邮件至 questions@packtpub.com。

勘误：尽管我们已经尽全力确保内容的准确性，但错误在所难免。如果你在本书中发现错误，请将此报告给我们，我们将不胜感激。请访问 www.packtpub.com/submit-errata，选择你的图书，单击勘误提交表单链接，然后输入详细信息。[①]

反盗版：如果你在互联网上以任何形式发现我们作品的任何非法复制品，请将网址或网站名称提供给我们，我们将不胜感激。请通过 copyright@packtpub.com 与我们联系并提供材料的链接。

成为作者：如果你有专业知识并且对撰写图书感兴趣，请访问 authors.packtpub.com。

① 本书中文版勘误请到http://ituring.cn/book/2396查看和提交。——编者注

评论

请留下评论。既然你阅读并使用了本书，为什么不在购买的网站上留下评论呢？这样，潜在读者就可以看到并根据你的客观意见做出购买决定，我们可以了解你对我们产品的看法，作者也可以看到你的反馈。谢谢！

有关 Packt 的更多信息，请访问 packtpub.com。

电子书

扫描如下二维码，即可购买本书中文电子版。

目　　录

第1章 入门

计算机视觉是理解或操作图像和视频的科学。计算机视觉有很多应用，包括自动驾驶、工业检验和增强现实。对于计算机视觉，深度学习的使用可以分为多个类别：图像和视频中的分类、检测、分割和生成。在本书中，我们将学习如何为计算机视觉应用训练深度学习模型，并将其部署在多个平台上。TensorFlow 将贯穿全书，这是一个流行的用于深度学习的 Python 库，我们将用它来实现示例。

本章涉及以下主题：

- 深度学习的基础知识和词汇；
- 深度学习与计算机视觉如何结合；
- 建立本书中示例所需的开发环境；
- 感受 TensorFlow 及其强大的工具，如 TensorBoard 和 TensorFlow Serving。

1.1 理解深度学习

计算机视觉是一个有着悠久历史的领域。随着深度学习的出现，计算机视觉已被证明可用于各种应用。深度学习集合了**人工神经网络**（artificial neural network，ANN）的技术，后者是机器学习的一个分支。人工神经网络以人脑作为模型的基础，其中有一些节点相互连接并相互传递信息。在下面几节中，我们将通过理解常用的基本术语来详细讨论深度学习的工作原理。

1.1.1 感知机

人工神经元又称感知机（perceptron），它接收若干输入并执行加权求和以产生输出。感知机的权重在训练过程中基于训练数据确定。图 1-1 是感知机的图示。

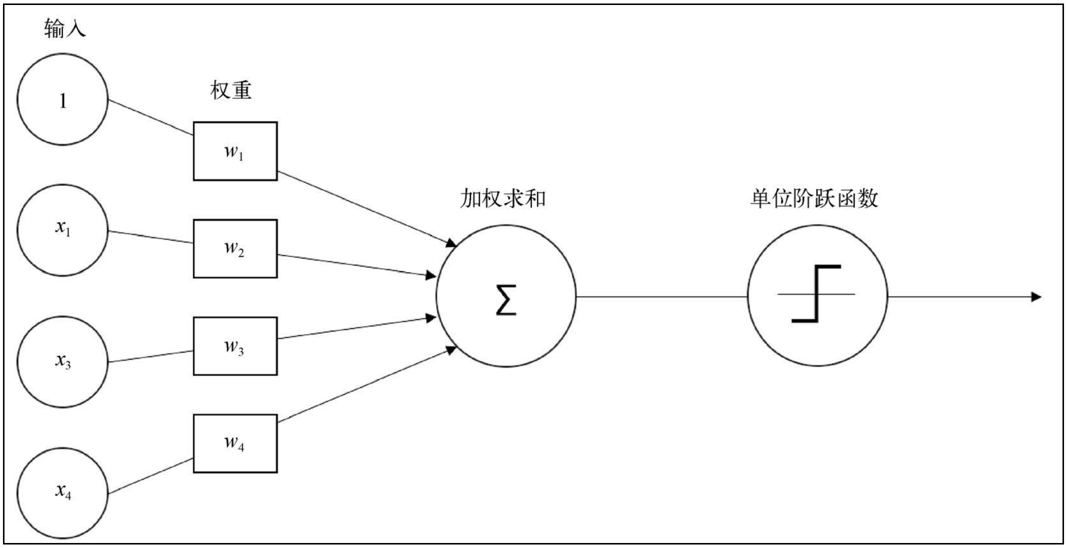


图　1-1

如图 1-1 所示，多个输入被加权并相加。在这个例子中，总和会经过一个单位阶跃函数用于二分类问题。该感知机只能通过从例子中学习权重来学习简单的功能。学习权重的过程称为训练。感知机的训练可以通过基于梯度的方法完成，1.1.4 节将详细解释。感知机的输出可以经过激活函数或转移函数来传递，这将在下一节中解释。

1.1.2　激活函数

激活函数（activation function）使**神经网络**具有非线性。它决定感知机是否应被激发。在训练期间，激活函数在调整梯度方面发挥着重要作用。激活函数能以更高的幅度衰减梯度值，如下面介绍的 `sigmoid`。激活函数这种非线性赋予了深度网络学习复杂函数的能力。除了在 0 点的修正单元，大多数激活函数都是连续函数和可微函数。连续函数输入中的每个小变化都会引起输出中的小变化。可微函数在定义域中的每个点都存在导数。

为了训练神经网络，激活函数必须是可微的。下面介绍一些激活函数。

TIP　如果你尚不明白“连续”和“可微”的意思，也不用担心。在随后的章节中，这些概念会变得更加清晰。

1. sigmoid

`sigmoid` 可以被看作平滑的阶跃函数，因此可微。`sigmoid` 可用于将任何值转换为概率，并可用于二分类问题。如图 1-2 所示，`sigmoid` 将输入映射到范围为 0 ~ 1 的值。

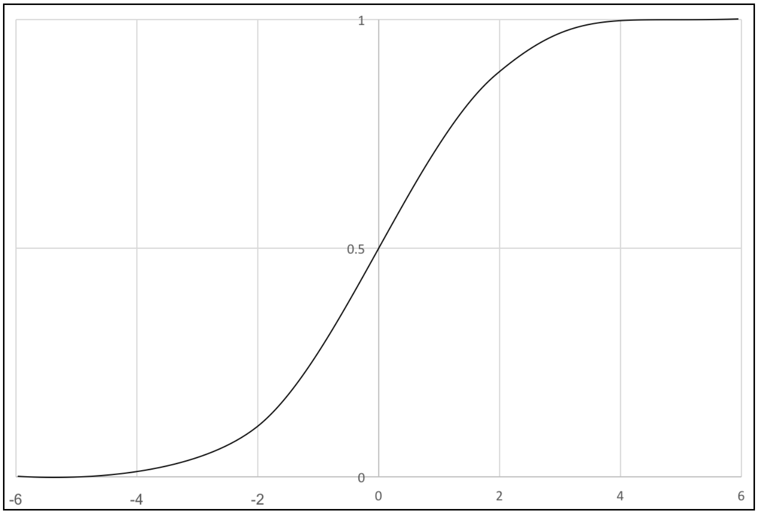


图　1-2

sigmoid 函数中，相对于 X 值的变化，Y 值的变化更小，因此将导致梯度逐渐消失。经过一番学习之后，变化可能非常小。下面将介绍另一个激活函数 tanh。它是 sigmoid 的缩放版本，可在一定程度上避免梯度值消失的问题。

2. 双曲正切函数

双曲正切函数 tanh 是 sigmoid 的缩放版本。与 sigmoid 一样，它是平滑和可微的。如图 1-3 所示，tanh 将输入映射到范围为 –1 ~ 1 的值。

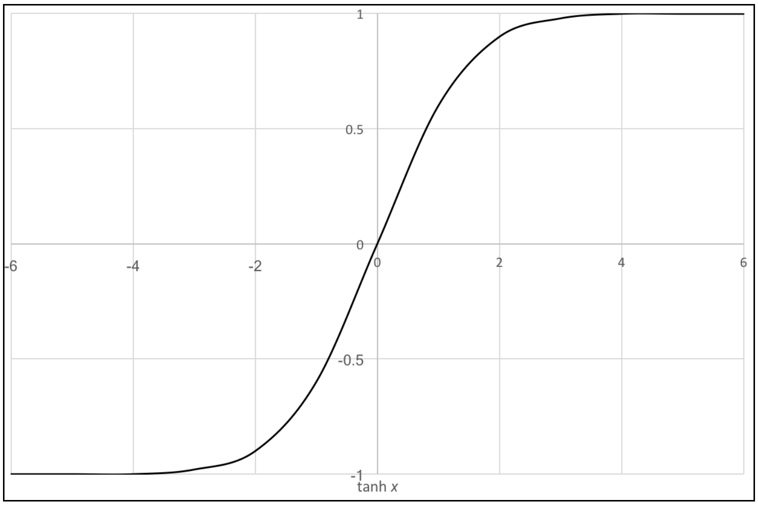


图　1-3

它的梯度比 `sigmoid` 更稳定，因此很少引起梯度消失问题。`sigmoid` 和 `tanh` 这两个激活函数都始终激发，使得相应的 ANN 很笨重。下面将引入另外一种激活函数，即修正线性单元（rectified linear unit，ReLU），它可以通过有时不激发来避免这个缺陷。

3. 修正线性单元

修正线性单元可以让大数值通过，而让一些神经元处于不会激发的陈旧态。这为神经网络增加了稀疏性这个好的特性。如图 1-4 所示，`ReLU` 将输入 x 映射到 $\max(0, x)$，即将负的输入映射为 0，而正的输入没有任何改变地输出。

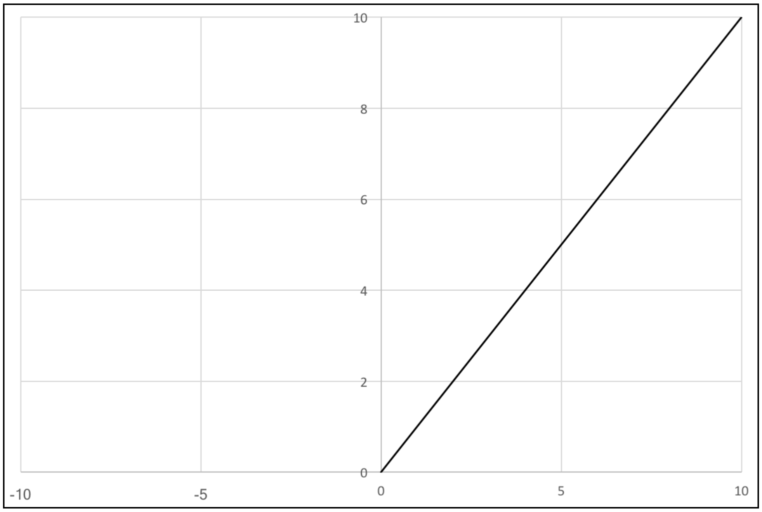


图　1-4

由于 `ReLU` 不是一直处在激发状态，因此可以加快训练速度。同时，由于该函数很简单，因此计算成本也最低。选择激活函数非常依赖于应用程序。尽管如此，`ReLU` 面对大多数问题时都可以良好地工作。在下一节中，我们将通过学习如何将多个感知机堆叠在一起来学习更复杂的函数。

1.1.3　人工神经网络

人工神经网络（ANN）是感知机和激活函数的集合。多个感知机相互连接，从而形成隐藏层或隐藏单元。隐藏单元形成非线性基础，将输入层映射到低维空间中的输出层，这也称为 ANN。ANN 是从输入到输出的映射。该映射通过用带偏差的输入加权来计算。这些权重值、偏置（bias）值以及架构统称为模型。

权重值和偏置值由训练过程确定。训练开始时，用随机值初始化模型值。误差是通过使用损失函数（loss function）比较观测值与真实值来计算的。根据计算出的损失，在每一步调整权重。当误差不能进一步减小时，就停止训练。训练过程在训练期间学习特征。这些特征比原始图像有

更好的表征。图 1-5 是人工神经网络（多层感知机）的图示。

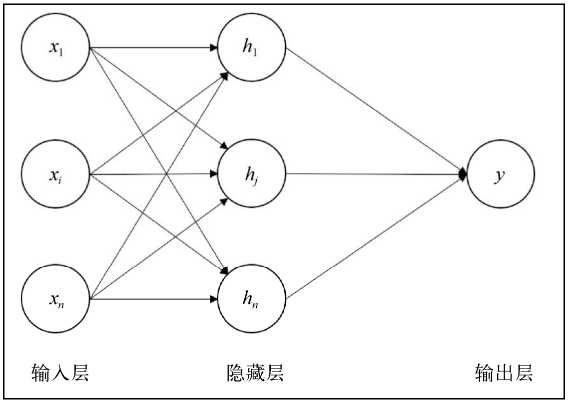


图 1-5

x 的几个输入通过感知机组成的隐藏层传递，并进行求和以得到输出。万能逼近定理（universal approximation theorem）表明：这样一个神经网络可以逼近任何函数。隐藏层也可以称为密集层。每一层都可以具有一个上一节中介绍的激活函数。隐藏层和感知机的数量可以根据具体问题来选择。还有一些因素使这种多层感知机适用于多分类问题。多分类问题可尝试区分十多个类别。下面将探讨这些术语。

1. 独热编码

在分类问题中，独热编码（one-hot encoding）是一种表示目标变量或类别的方法。目标变量可以由字符串标签转换为独热编码向量。一个独热向量在目标类别的索引处填充 1，在其他地方填充 0。例如，如果目标类别是猫和狗，它们可以分别用[1, 0]和[0, 1]表示。对于 1000 个类别，独热编码向量的大小为 1000 个整数，其中除一个数为 1 外全为 0。独热编码不对目标变量的相似性做假设。下面将解释独热编码与 `softmax` 的组合，而这使 ANN 多分类成为可能。

2. softmax

`softmax` 是迫使神经网络输出总和为 1 的一种方法。因此，`softmax` 函数的输出值可以视为概率分布的一部分。这在多分类问题中很有用。`softmax` 是一种特殊的激活函数，其输出总和为 1。它通过将输出除以所有值的总和将输出转换为概率。可以通过计算 `softmax` 概率和独热编码之间的欧几里得距离来解决优化问题。但用于优化问题的一个更好的代价函数是下面的交叉熵。

3. 交叉熵

交叉熵（cross-entropy）可用于比较 `softmax` 输出和独热编码输出之间的距离。交叉熵是一种损失函数，其误差必须最小化。神经网络估计给定数据映射到每个类别的概率。所映射的概率必须在正确的目标标签上取得最大值。交叉熵是负对数概率的总和。这里使用对数是为了提高数

字值的稳定性。最大化函数值等同于最小化相同函数的负值。下面将介绍以下避免 ANN 过拟合的正则化方法。

- 丢弃
- 批归一化
- L1 和 L2 正则化

4. 丢弃

丢弃（dropout）是避免过拟合 ANN 的一种正则化神经网络的有效方法。如图 1-6 所示，在训练期间，丢弃层通过随机移除部分隐藏单元来消减神经网络。

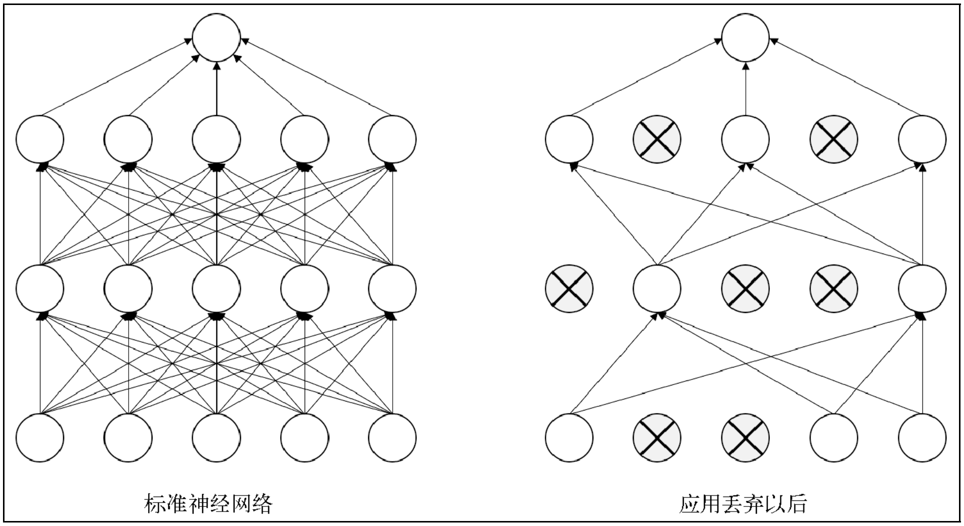


图　1-6

注意神经元是如何随机训练的。丢弃也是组合多个神经网络的有效方式。对于每个训练案例，我们随机选择一些隐藏单元，以便针对每个案例得到不同的结构。这是装袋（bagging）和模型平均（model averaging）的极端情况。但是，在推断过程中，不应使用丢弃层，因为没有必要。

5. 批归一化

批归一化（batch normalization，或称 batch-norm）提高了神经网络训练的稳定性和性能。它对一层的输出进行零均值和标准差为 1 的归一化。这可以减少过拟合，并使网络训练更快。它在训练复杂神经网络中非常有用。

6. L1 和 L2 正则化

L1 惩罚权重的绝对值并使权重趋于为零。L2 惩罚权重的平方值，并在训练期间倾向于使权重更小。这两种正则化方法都假设权重较小的模型更好。

1.1.4 训练神经网络

因为 ANN 包含若干个参数要优化，所以训练它颇有难度。更新权重的过程称为反向传播（backpropagation）。最小化错误的过程称为优化。下面将详细介绍它们。

1. 反向传播

反向传播算法通常用于训练 ANN。如图 1-7 所示，该算法根据计算出的误差向后更新权重。

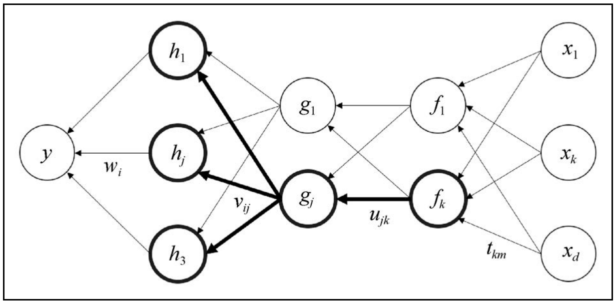


图 1-7

计算出误差后，可以使用梯度下降计算权重更新，如下所述。

2. 梯度下降

梯度下降算法执行多维度优化。目标是达到全局最大值。梯度下降是许多机器学习模型中流行的优化技术。它用于改进或优化模型预测。梯度下降的一种实现称为**随机梯度下降**（stochastic gradient descent，SGD），它在神经网络中变得越来越流行。优化包括计算误差值并更新权重以实现最小误差。找到最小值的方向是损失函数梯度的负值。图 1-8 定性地显示了梯度下降过程。

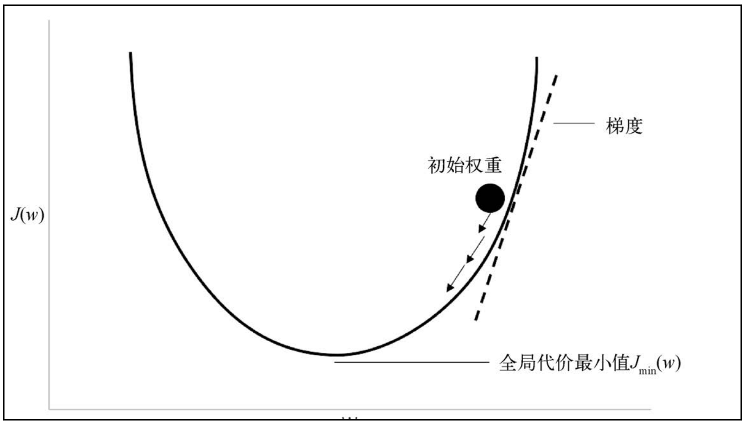


图 1-8

学习速率决定了每个步骤的大小。请注意，具有非线性激活函数的 ANN 将具有局部最小值。SGD 在实践上可以更好地优化非凸的代价函数。

3. 随机梯度下降

除了每次仅用部分样本数据进行训练以外，SGD 与梯度下降相同。相应的参数称为小批次规模（mini-batch size）。理论上，即使只有一个样本也可以用于训练。在实践中，最好尝试不同个数的样本数据。在下一节中，我们将讨论卷积神经网络，它能比标准 ANN 更好地处理图像数据。

1.1.5　尝试 TensorFlow 游乐场

TensorFlow 游乐场（playground）是神经网络的交互式可视化平台。访问 playground.tensorflow.org 网站，通过更改参数来查看先前提到的术语是如何协同工作的。图 1-9 是 TensorFlow 游乐场的截图。

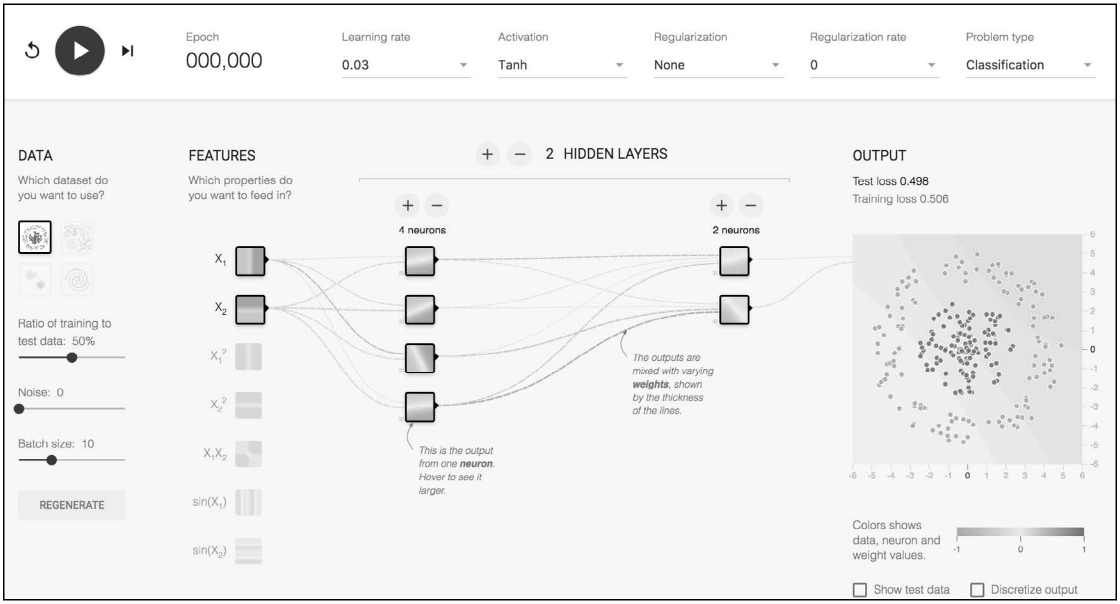


图　1-9

如图 1-9 所示，可以改变学习速率、激活函数、正则化、隐藏单元和各个层，看看会如何影响训练过程。我们可以花一些时间调整参数，以直观地了解神经网络是怎样处理各种数据的。

1.1.6　卷积神经网络

卷积神经网络（convolutional neural network，CNN）类似于前面描述的神经网络。CNN 具

有权重、偏置项和经过非线性激活函数的输出。一般的神经网络接收输入后，神经元全连接到下一层。同一层内的神经元不共享任何连接。如果对图像使用一般的神经网络，由于神经元数量庞大，它们的规模会非常大，导致过拟合。因为图像规模很大，所以不能将这种神经网络直接用于图像。增加模型的规模需要大量的神经元。可以认为图像是具有高度、宽度和深度尺寸的容积（volume）。深度是图像的通道，即红色、蓝色和绿色。CNN 的神经元以容积方式排列以利用容积的优势。每个层都将输入容积转换为输出容积，如图 1-10 所示。

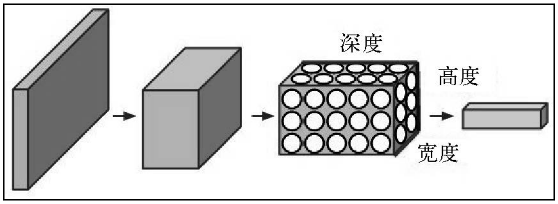


图 1-10

卷积神经网络通过变换对编码滤波。学习到的滤波器可以检测图像中的特征或模式。层越深，模式越抽象。一些分析表明，这些层具有检测边缘、角落和模式的能力。CNN 各层中的可学习参数比上一节中描述的密集层要少。

1. 核

核（kernel）是用于卷积图像的参数卷积层。卷积操作如图 1-11 所示。

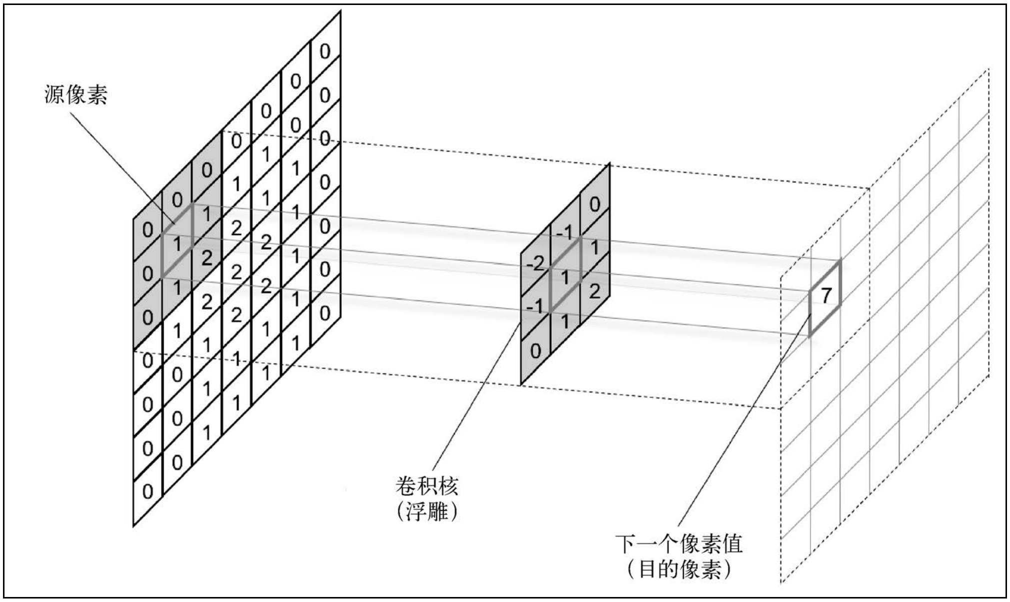


图 1-11

核有两个参数，称为步幅（stride）和尺寸（size）。尺寸可以是矩形的任何一维。步幅是每次移动的像素数量。步幅长度为1时生成几乎相同尺寸的图像，而步幅长度为2时生成的图像只有原本尺寸的一半。额外填充（padding）图像将有助于实现相同尺寸的输入。

2. 最大池化

池化层位于卷积层之间。池化层通过采样在层间减小图像大小。最大池化（max pooling）的采样通过在窗口中选择最大值来完成。平均池化（average pooling）对窗口内的值做平均运算。池化也是一种正则化技术，可以避免过拟合。池化在特征的所有通道上进行。池化也可以以不同的步幅进行。

窗口的大小是CNN感受野（receptive field）的一个度量。图1-12显示了最大池化的示例。

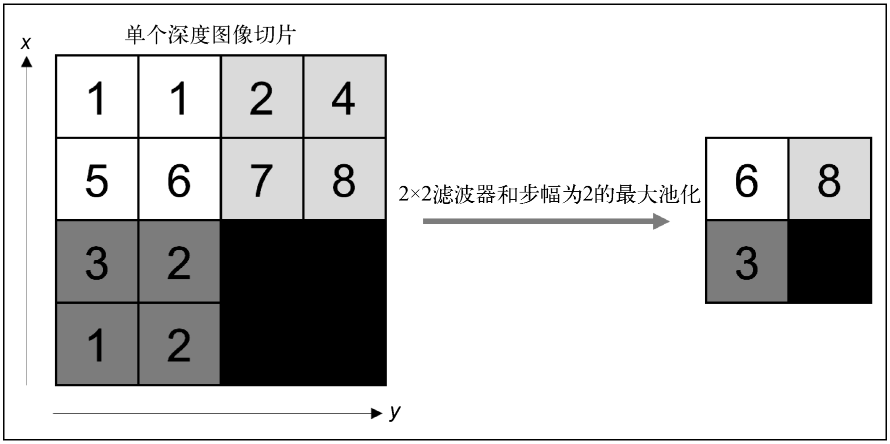


图 1-12

CNN是所有计算机视觉深度学习模型中最重要的组成部分。毫不夸张地说，没有CNN，任何计算机都不可能有视觉。在接下来的几节中，我们将讨论几个可用于一些应用的先进层设计。

说明：可访问 https://www.youtube.com/watch?v=jajksuQW4mc，了解CNN和最大池化操作的可视化效果。

1.1.7 循环神经网络

循环神经网络（recurrent neural network，RNN）可对序列信息建模。它不假设数据量很大。它从一系列序列数据的先前数据输出中执行相同的任务。这也可以认为是记忆。RNN无法记住更长的序列或时间。它在训练过程中展开，如图1-13所示。

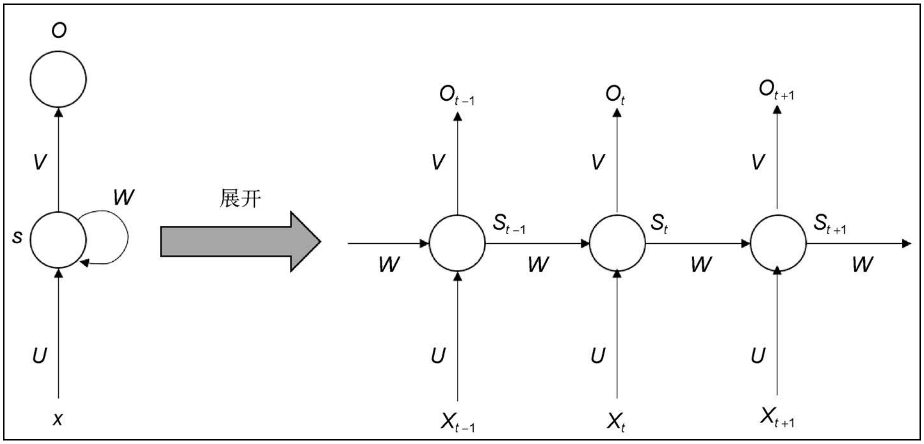


图 1-13

由上图可知，其操作步骤每次都要展开和训练。在反向传播过程中，随着时间的推移，梯度会消失。为了克服这个问题，可以使用**长短期记忆网络**（long short-term memory，LSTM）来记住更长的时间段。

1.1.8 长短期记忆网络

长短期记忆网络可以存储更长时间的信息，因此它可以高效地获取长期效率。图 1-14 说明了 LSTM 单元（cell）的设计。

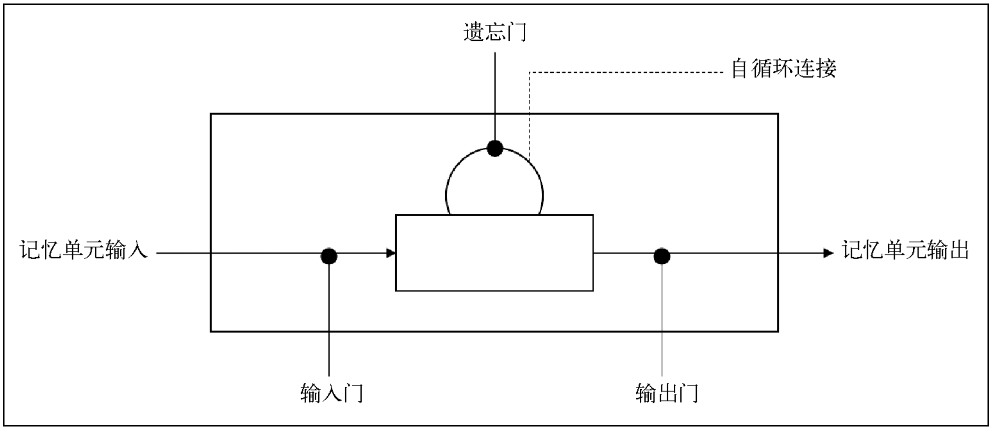


图 1-14

LSTM 有几个门：遗忘门、输入门和输出门。遗忘门保持了以前的状态信息。输入门使用输入更新当前状态。输出门决定信息传递到下一个状态。遗忘和保留重要事物的能力使 LSTM 能够

在更长的时间内记住信息。

你已经学会了贯穿整本书的深度学习术语。下一节将介绍如何在计算机视觉的背景下使用深度学习。

1.2　计算机视觉深度学习

计算机视觉能够使计算机具有人类视觉特性。计算机的形式可以是智能手机、无人机、闭路电视、磁共振成像扫描仪等，它们具有用于感知的各种传感器。传感器以数字形式生成图像，然后由计算机进行解释。下一节将介绍这种智能的基本组成部分。使用深度学习技术可以有效解决计算机视觉中出现的不同问题。

1.2.1　分类

图像分类是以一定的可信度用一个对象或概念对整幅图像打标签的任务。它的应用包括根据给定人脸图像进行性别分类、识别宠物类型、标记照片等。图 1-15 是这种分类任务的一个输出。

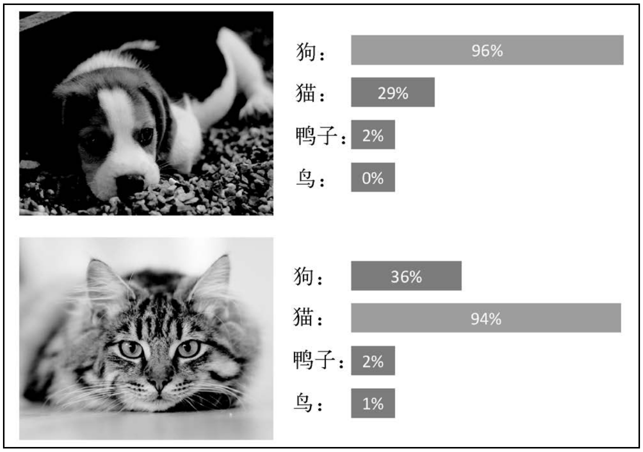


图　1-15

第 2 章将详细介绍可用于分类任务的方法；在第 3 章中，我们将使用深度学习模型的可视化分类模型并检索出相似的图像。

1.2.2　检测或定位与分割

检测或定位是在图像中查找目标并使用边界框对目标进行定位的一项任务。这项任务有许多应用，例如为自动驾驶车辆寻找行人和标识牌。图 1-16 是检测的一个图示。

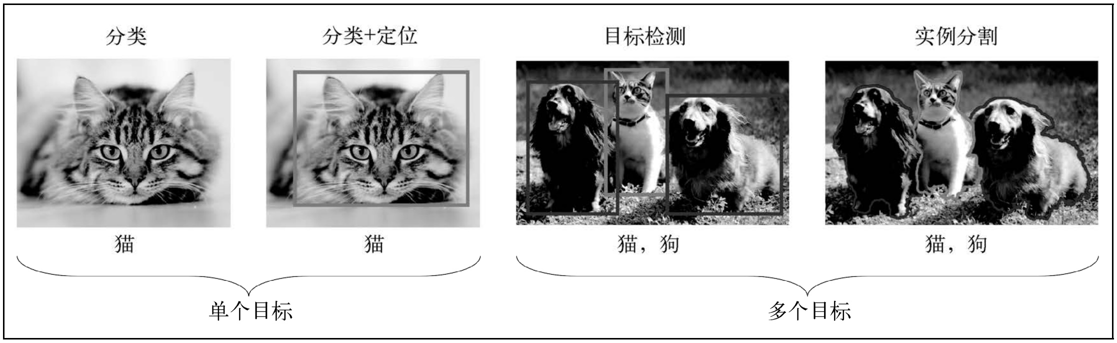


图　1-16

分割是对图像进行像素级分类的任务。它可以精细化分离对象。这对处理医学图像和卫星图像非常有用。可以在第 4 章和第 5 章中找到更多示例和解释。

1.2.3　相似性学习

相似性学习（similarity learning）是学习两幅图像相似程度的过程。可以基于语义计算两幅图像之间相似度的评分，如图 1-17 所示。

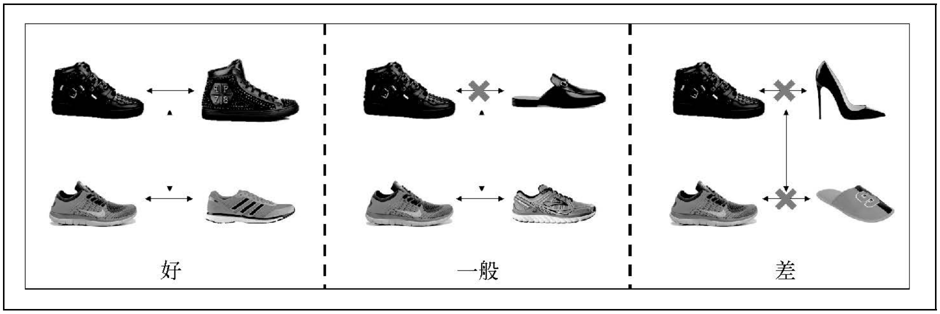


图　1-17

它有一些应用，包括找到类似的产品以及执行人脸识别。第 6 章将介绍相似性学习技术。

1.2.4　图像题注

图像题注（image captioning）是用文本描述图像的任务，如图 1-18 所示。



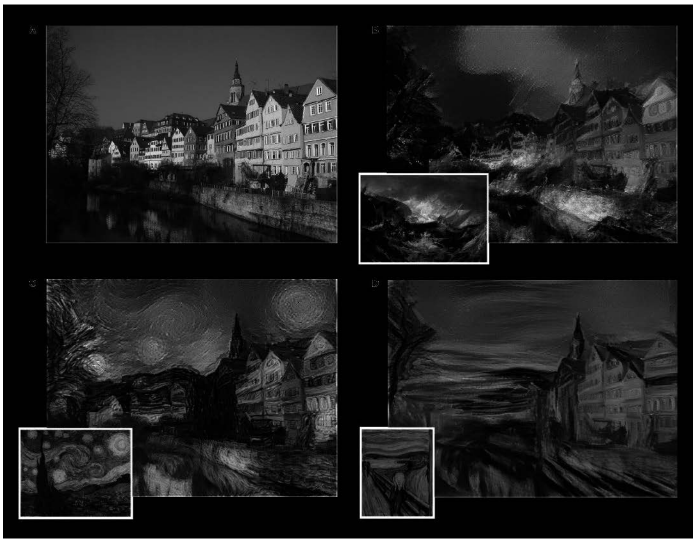
* 经 Vinyals 等人许可转载

图　1-18

第 7 章详细介绍了图像题注技术。这是一个将**自然语言处理**（natural language processing，NLP）技术和计算机视觉技术相结合的独特案例。

1.2.5　生成模型

生成模型（generative model）在生成图像时非常有趣。图 1-19 是风格迁移应用的一个示例，其中使用一幅图像的内容融合其他图像的风格生成新的图像。



* 经 Gatys 等人许可转载

图　1-19

生成图像也有其他目的，例如新的训练样本、超分辨率图像等。第 8 章将详细介绍生成模型技术。

1.2.6　视频分析

视频分析从整体上处理视频，而不是像以前的例子中那样处理图像。它有一些应用，如运动追踪、入侵检测和监控摄像头。第 9 章将讨论特定于视频的应用。时间数据的新维度带来了许多有趣的应用。

在下一节中，我们将学习如何建立开发环境。

1.3　建立开发环境

本节我们将建立编程环境，这对于本书其余部分中的示例非常有用。我们可以选择以下操作系统。

- 开发用操作系统（OS）：如 Mac、Ubuntu 或 Windows。
- 部署用操作系统：如 Mac、Windows、Android、iOS，或 Amazon Web Services（AWS）、Google Cloud Platform（GCP）、Azure 等云平台上安装的 Ubuntu、Tegra、Raspberry Pi。

不管哪种平台，本书中开发的所有代码都应该能顺利运行。在本章中，我们将介绍开发环境的安装过程。第 10 章将介绍在 AWS、GCP、Azure、Tegra 和 Raspberry Pi 等各种环境中部署的安装。

1.3.1　硬件和操作系统

对于开发环境而言，由于训练的计算成本非常高，因此需要具备大量计算能力。Mac 用户拥有的计算能力相当有限。Windows 和 Ubuntu 用户可以使用更多的处理器和通用图形处理器（general purpose-graphics processing unit，GP-GPU）来加强他们的开发环境，这将在下一节中解释。

通用图形处理器

GP-GPU 是可以加速深度学习模型训练过程的特殊硬件。因为有成熟的软件和社区支持，所以 NVIDIA 公司提供的 GP-GPU 在深度学习训练和部署方面非常受欢迎。我们可以使用这种 GP-GPU 机器加快训练速度。我们有很多选择，可以根据预算选择其中一种。选择与 GP-GPU 功率匹配的 RAM、CPU 和硬盘也很重要。安装硬件后，必须安装以下驱动程序和库。使用 Mac 或使用没有 GP-GPU 的 Windows/Ubuntu 的读者可以跳过下述安装。

以下是建立环境所需的库：

- ❑ CUDA
- ❑ CUDNN

- **CUDA**

CUDA（computer unified device architecture，计算统一设备架构）是 NVIDIA 提供的 API 层，它使用 GPU 的并行特性。安装 CUDA 后，还会安装硬件驱动程序。首先，从 NVIDIA 网站下载 CUDA 库。

阅读页面上的说明，下载驱动程序，然后按照安装说明进行操作。图 1-20 是 Ubuntu CUDA 和安装说明的屏幕截图。

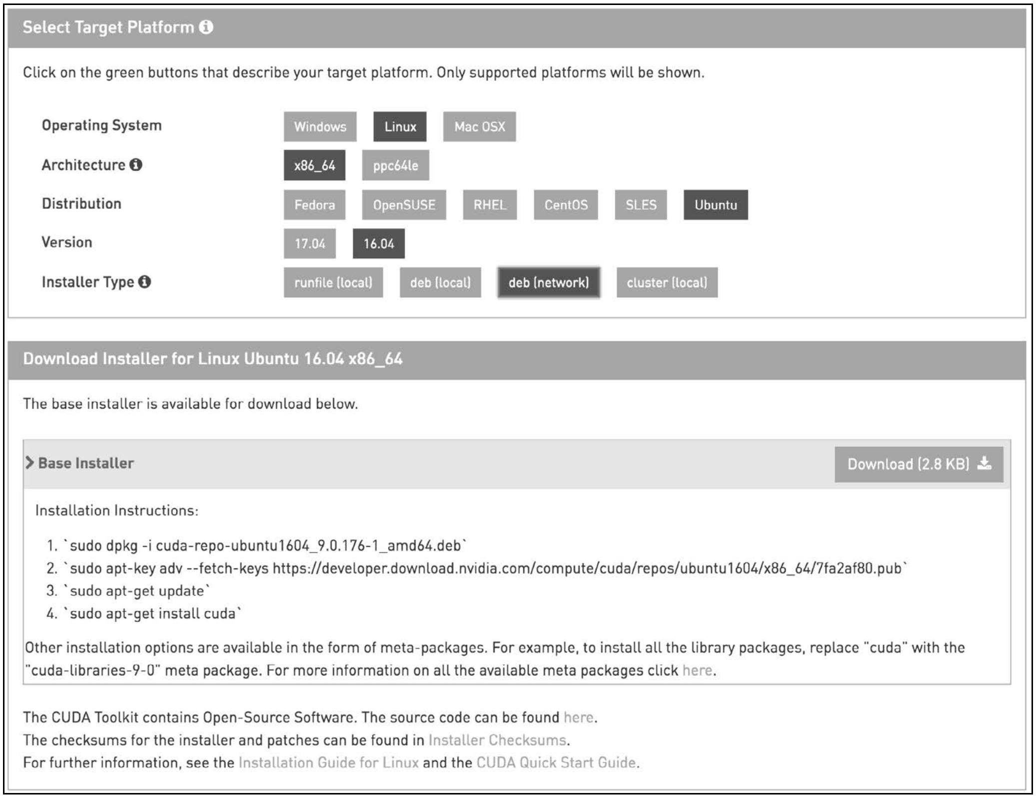


图　1-20

这些命令将安装需要的 cuda 驱动程序和其他 CUDA API。

> TIP：可以通过在命令提示符下键入 `nvidia-smi`，来检查驱动程序是否安装正确。

- **CUDNN**

CUDNN（CUDA deep neural network）库为深度学习算法提供原语。这个软件包是由 NVIDIA 提供的，因此它的硬件经过高度优化，运行速度更快。软件包中提供了几个标准的深度学习例程。这些软件包由著名的深度学习库（如 TensorFlow、Caffe 等）使用。在下一节中，提供了有关安装 CUDNN 的说明。你可以从 NVIDIA 网站下载 CUDNN。

> 用户账户是必需的（免费注册）。

将相关文件复制到 CUDA 文件夹，使其更快地在 GPU 上运行。我们不会直接使用 CUDA 和 CUDNN 库。TensorFlow 使用优化例程让它们在 GP-GPU 上工作。

1.3.2 安装软件包

训练深度学习模型需要几个库。我们将安装以下库并介绍选择以下软件包的原因。

- Python 和其他依赖
- OpenCV
- TensorFlow
- Keras

1. Python

Python 是任何数据科学应用事实上的选择。它拥有最大的社区和库的支持生态系统。用于 Python 的 TensorFlow API 是最完整的，因此 Python 是编程语言的首选。Python 有两个版本：Python 2.*x* 和 Python 3.*x*。本书中将讨论 Python 3.*x*。这个选择有以下三个原因：

- Python 2.*x* 的开发将在 2020 年停止，因此 Python 3.*x* 是 Python 的未来；
- Python 3.*x* 避免了原始实现中的许多设计缺陷；
- 与普遍的看法相反，Python 3.*x* 拥有与 Python 2.*x* 一样多的数据科学支持库。

我们将在本书中使用 Python 版本 3。根据操作系统，转到 python.org 网站并下载版本 3。按照下载链接中给出的步骤安装 Python。安装 Python 之后，必须安装 pip3 以便安装 Python 包。然后通过输入以下命令安装几个 Python 软件包，就可以安装 OpenCV 和 TensorFlow 了。

```
sudo pip3 install numpy scipy scikit-learn pillow h5py
```

前面安装的软件包的描述如下。

- NumPy 是一个高度优化的数值计算软件包。它具有强大的 *N* 维数组对象包，而且 NumPy 库的矩阵运算对速度进行了高度优化。图像可以存储为三维的 `numpy` 对象。

- ❑ SciPy 有几个用于科学和工程计算的例程。本书后面将使用它的一些优化包。
- ❑ scikit-learn 是一个机器学习库，我们将使用它的许多辅助函数。
- ❑ Pillow 对图像加载和基本操作非常有用。
- ❑ H5py 包是 HDF5 二进制数据格式的 Python 接口。这是存储使用 Keras 训练模型的格式。

2. OpenCV

OpenCV（open computer vision）是一个著名的计算机视觉库。这个库中有一些非常有用的图像处理例程。以下是在 Ubuntu 中安装 OpenCV 的步骤：

```
sudo apt-get install python-opencv
```

可以在 https://opencv.org/上找到其他操作系统安装的相似步骤。OpenCV 是跨平台的，并针对 CPU 密集型应用进行了优化。它具有多种编程语言的接口，在 Windows、Ubuntu 和 Mac 上都有支持。

3. TensorFlow 库

TensorFlow 是开发和部署深度学习模型的开源库。TensorFlow 使用计算图进行数据流和数值计算。换句话说，数据或张量（tensor）在图中流动，因此称为 TensorFlow。该图具有可以进行任何数值计算的节点，因此适用于深度学习操作。它为各种平台和硬件提供一致的 API。TensorFlow 在后端处理规模化和优化的所有复杂性。它最初是为 Google 的研究而开发的。它是最著名的深度学习库，拥有大型社区，并配有可视化和生产部署工具。

- **安装 TensorFlow**

使用以下命令，为 CPU 使用 pip3 安装 TensorFlow：

```
sudo pip3 install tensorflow
```

如果你正在使用 GPU 硬件并安装了 CUDA 和 CUDNN，可使用以下命令安装 TensorFlow 的 GPU 版本：

```
sudo pip3 install tensorflow-gpu
```

现在 TensorFlow 已经安装完成并可以使用。我们通过几个例子来了解 TensorFlow 是如何工作的。

- **打印“Hello, TensorFlow”的 TensorFlow 例子**

我们将直接在 Python shell 中使用 TensorFlow 来举一个例子。在这个例子中，我们将使用 TensorFlow 打印 Hello, TensorFlow。

(1) 在命令提示符下键入以下命令从 shell 中调用 Python。

```
python3
```

(2) 输入以下命令以导入 TensorFlow 库。

```
>>> import tensorflow as tf
```

(3) 接下来，用字符串 Hello, TensorFlow 定义一个常量。这与通常的 Python 赋值操作不同，因为该值尚未初始化。

```
>>> hello = tf.constant('Hello, TensorFlow!')
```

(4) 创建一个会话来初始化计算图，并为会话（session）命名。

```
>>> session = tf.Session()
```

会话可以使用变量 `hello` 作为参数运行。

(5) 现在计算图可执行并返回打印的特定变量。

```
>>> print(session.run(hello))
```

它应该打印以下内容：

```
Hello, TensorFlow!
```

让我们再看一个例子，理解会话和计算图是如何工作的。

访问 https://github.com/rajacheers/DeepLearningForComputerVision，获取书中所有示例的代码。代码按照章节进行组织。你可以提出问题并在存储库中获得帮助。

- **相加两个数字的 TensorFlow 例子**

这是 TensorFlow 相加两个数字的一个简单示例。

(1) 使用以下代码创建一个 Python 文件并导入 TensorFlow。

```
import tensorflow as tf
```

前面的导入对于后面的所有例子都是必需的。假设我们已经为所有示例导入了库。占位符（placeholder）可以通过以下方式定义。分配时未加载占位符。这里，变量被定义为一个 `float32` 类型的占位符。占位符是一个空的声明，并且可以在会话运行时取值。

(2) 定义一个占位符，如下所示。

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
```

(3) 现在可以将占位符的总和操作定义为通常的求和。这里，该操作不会执行，而只是使用以下代码进行定义。

```
z = x + y
```

(4) 可以按照前面的示例所示创建会话。如下定义后，计算图就准备好执行计算了。

```
session = tf.Session()
```

(5) 以字典格式定义占位符的值。

```
values = {x: 5.0, y: 4.0}
```

(6) 使用变量 c 和定义的值运行会话。计算图将值提供给适当的占位符，并将值返回给变量 c。

```
result = session.run([z], values)
print(result)
```

作为相加的结果，该程序应打印[9.0]。

可以理解的是，这不是相加两个数字的最佳方式。这个例子是为了理解 TensorFlow 中张量和操作的定义。想象一下，使用万亿个数字并将其相加是多么困难。TensorFlow 使用相同的 API 轻松地实现了这样规模的计算。下面将介绍如何安装和使用 TensorBoard 和 TensorFlow Serving。

- **TensorBoard**

TensorBoard 是一套可视化工具，用于使用 TensorFlow 对基于深度学习的模型进行训练。以下数据可以在 TensorBoard 中可视化。

- ❑ 图：计算图、设备布局和张量详细信息。
- ❑ 标量：诸如损失、迭代的准确率等度量指标。
- ❑ 图像：用于查看带有相应标签的图像。
- ❑ 音频：用于收听训练或生成的音频。
- ❑ 分布：用于查看某个标量的分布。
- ❑ 直方图：包含权重和偏置项的直方图。
- ❑ 投影器：帮助在三维空间中可视化数据。
- ❑ 文本：打印训练文本数据。
- ❑ 简况：查看用于训练的硬件资源。

TensorBoard 与 TensorFlow 一起安装。类似于前面的示例，转到 Python 3 提示符并键入以下命令以开始使用 TensorBoard：

```
x = tf.placeholder(tf.float32, name='x')
y = tf.placeholder(tf.float32, name='y')
z = tf.add(x, y, name='sum')
```

请注意，参数名称已作为占位符和操作的额外参数提供。这些都是可视化图形时可以看到的名称。现在可以使用 TensorBoard 中的以下命令将图形写入特定文件夹：

```
session = tf.Session()
summary_writer = tf.summary.FileWriter('/tmp/1', session.graph)
```

该命令将图形写入磁盘上参数中给定的文件夹。现在可以用以下命令调用 TensorBoard：

```
tensorboard --logdir=/tmp/1
```

任何目录都可以作为存储文件的 `logdir` 选项的参数传递。转到浏览器并粘贴以下 URL，以启动可视化来访问 TensorBoard。

http://localhost:6006/

浏览器窗口中的 TensorBoard 可视化如图 1-21 所示。

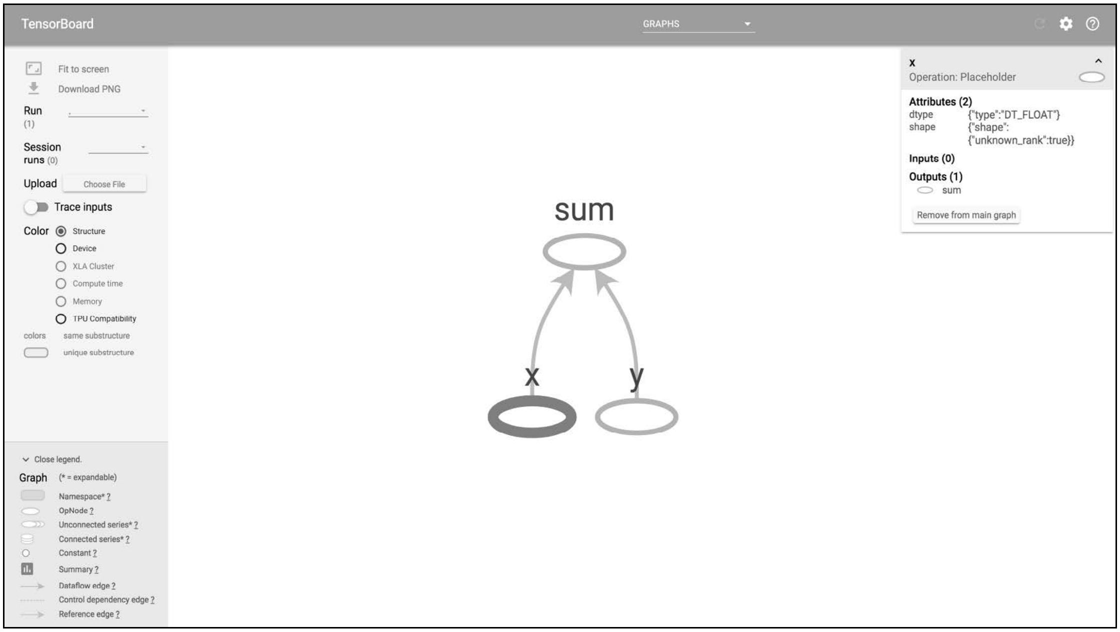


图 1-21

求和的图显示有占位符的名称。当点击它们时，可以在右侧看到该操作张量的所有细节。我们要让自己熟悉操作的选项卡和选项。这个窗口有几个部分，我们将在不同的章节中了解它们。TensorBoard 是使 TensorFlow 脱颖而出的最佳工具之一。

- **TensorFlow Serving 工具**

TensorFlow Serving 是 TensorFlow 中用于灵活部署环境开发、提供高延迟和吞吐量环境的一

款工具。任何使用 TensorFlow 训练的深度学习模型都可以通过 Serving 进行部署。通过运行以下命令来安装 Serving：

```
sudo apt-get install tensorflow-model-server
```

有关如何使用它的分步说明将在第 3 章中介绍。注意，在 Ubuntu 中安装 TensorFlow Serving 比较容易，其他操作系统请参阅 tensorflow.org 网站。图 1-22 说明了 TensorFlow Serving 和 TensorFlow 如何在生产环境中进行交互。

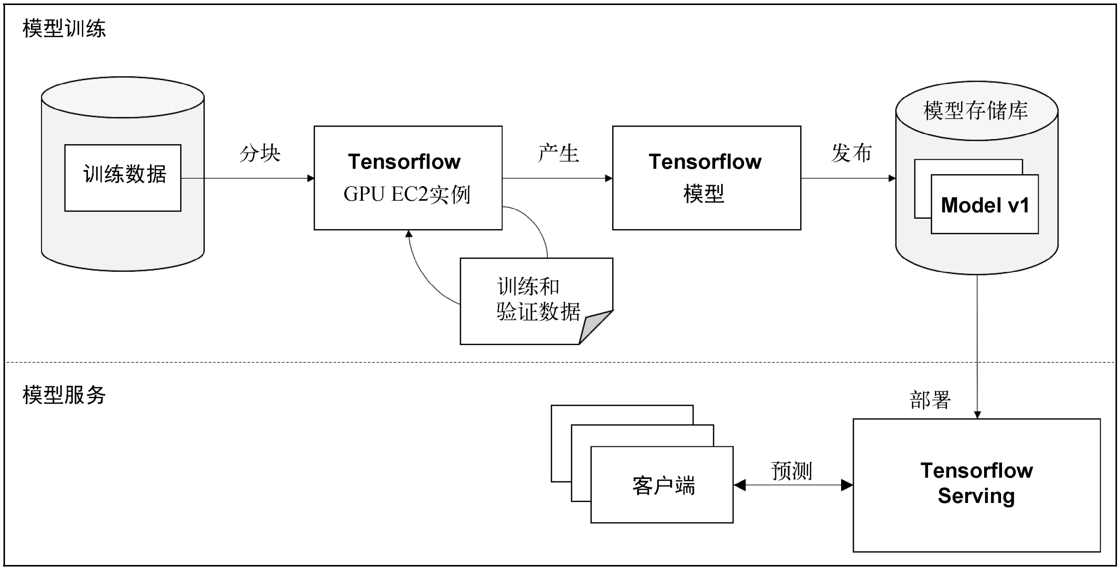


图　1-22

训练过程中可以生成许多模型，Serving 可以无缝地切换它们，而无须停机。除了第 3 章和第 10 章之外，所有后续章节都不需要使用 TensorFlow Serving。

4. Keras 库

Keras 是一个用 Python 编写的深度学习开源库。它提供了一个简单的界面来使用 TensorFlow 后端。Keras 也可以与 Theano、deep learning 4j 或 CNTK 一起使用，将它们用作后端。Keras 旨在通过专注于友好、模块化和可扩展性来轻松、快速地进行实验。它是一个独立的框架，可以在 CPU 和 GPU 之间无缝运行。Keras 可以单独安装，也可以使用 `tf.keras` API 在 TensorFlow 内部使用。在本书中，我们将使用 `tf.keras` API。我们已经介绍了为开发环境安装所需库的步骤。安装好并平稳运行 CUDA、CUDNN、OpenCV、TensorFlow 和 Keras 对于后面的章节至关重要。

1.4 小结

本章介绍了深度学习的基础知识。本章介绍的词汇将贯穿全书，因此，你可以经常回顾本章内容。本章通过示例展示了计算机视觉的一些应用，还介绍了创建开发环境时各种平台上所有软件包的安装。

下一章将讨论如何在数据集上使用 Keras 和 TensorFlow 来训练分类模型。我们将研究如何使用更大的模型和其他技术（如增强和微调）来提高准确性，还会介绍世界各地的人们提出的几个先进模型，它们在竞赛中实现了最好的准确性。

第 2 章

图像分类

图像分类是将整幅图像分类为单个标签的任务。例如，如果给定的图像是狗或猫，图像分类任务就可以将图像标记为狗或猫。本章将介绍如何使用 TensorFlow 来建立这样的图像分类模型，并学习提高分类准确率的技术。

本章涉及以下主题：

- 在 TensorFlow 中训练 MNIST 模型；
- 在 Keras 中训练 MNIST 模型；
- 其他流行的图像测试数据集；
- 更大的深度学习模型；
- 训练猫与狗的模型；
- 开发现实世界的应用。

2.1 在 TensorFlow 中训练 MNIST 模型

在本节中，我们将了解 MNIST（modified national institute of standards and technology）数据库的数据并建立一个简单的分类模型。本节的目标是学习深度学习的通用框架，并使用 TensorFlow 建立它。首先，建立一个感知机或逻辑回归模型。然后，训练一个 CNN 以获得更好的准确性。我们还将看到 TensorBoard 是如何帮助可视化训练过程并理解参数的。

2.1.1 MNIST 数据集

MNIST 数据中有手写数字 0 ~ 9，其中 60 000 幅图像用于训练，10 000 幅图像用于测试。该数据库被广泛用于尝试最少预处理的算法。它是学习机器学习算法的一个很好且简洁的数据库，也是最著名的面向图像分类问题的数据库。图 2-1 显示了一些示例。

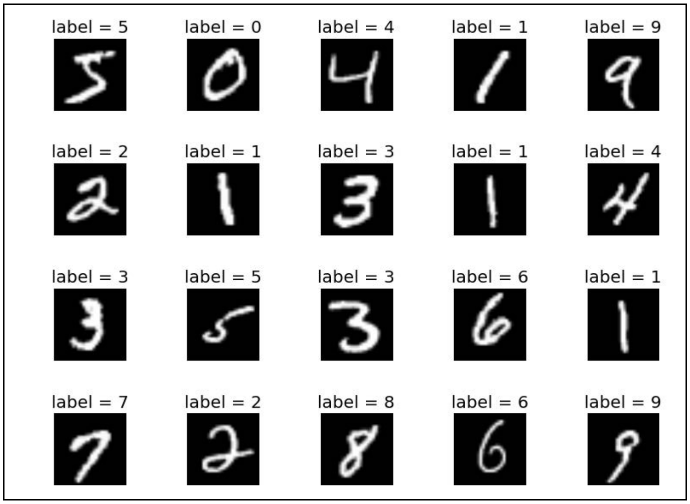


图 2-1

从上图可以看出，这些手写字符有 10 个标签。这些图像都被归一化为 28 像素 × 28 像素大小，转换为灰度值并且以固定大小居中。这是一个可以快速测试算法的小数据集。在下一节中，我们将看到如何加载此数据集以用于 TensorFlow。

2.1.2 加载 MNIST 数据

直接从 TensorFlow 加载 MNIST 数据。请注意，在加载数据时将独热编码指定为参数。标签以整数形式存储，但应作为独热编码加载以便于训练。假设读者从此时起用 TensorFlow 导入的 `tf` 从编辑器运行代码。以下是加载 `MNIST_data` 的代码片段：

```
from tensorflow.examples.tutorials.mnist import input_data
mnist_data = input_data.read_data_sets('MNIST_data', one_hot=True)
```

第一次运行时，下载数据可能需要一些时间。从第二次运行开始，将使用缓存的数据。在下一节中，我们将建立一个感知机对数字进行分类。

2.1.3 建立一个感知机

感知机是单层的神经网络。本章涵盖的概念，如全连接层、激活函数、随机梯度下降、`logits`、独热编码、`softmax` 和交叉熵，在这里会很有用。下面将学习如何在 TensorFlow 中定义神经网络的这些组件并使用网络来训练 MNIST 数据。

1. 为输入数据和目标定义占位符

占位符是传递数据的张量。占位符不是特定的值，但会在计算过程中接收输入。感知机的输

入大小、分类数量、批次大小以及迭代次数或批次总数要先声明。x_input 是稍后传入图像的输入占位符，y_input 是一次性标签或目标的占位符，如下所示：

```
input_size = 784
no_classes = 10
batch_size = 100
total_batches = 200

x_input = tf.placeholder(tf.float32, shape=[None, input_size])
y_input = tf.placeholder(tf.float32, shape=[None, no_classes])
```

因为还没有定义批次大小，所以 shape 参数中的 None 表示它可以是任何大小。第二个参数是 x_input 的张量大小和 y_input 的分类数量。根据占位符的类型，我们已将数据作为浮点数发送。接下来，我们可以定义感知机。

2. 定义全连接层的变量

我们通过解释权重和偏置等变量来定义一个简单的线性分类器或感知机。这些变量的值将在计算过程中习得。它们也被称为模型参数。权重变量用正态随机分布初始化，具有输入大小和分类数量的形状。当图像被重整形为一个单个向量时，输入大小为 784。分类的数量是 10，等于数据集中的数字个数。偏置变量也用随机正态分布初始化，其大小等于分类的数量。权重和偏置的定义如下：

```
weights = tf.Variable(tf.random_normal([input_size, no_classes]))
bias = tf.Variable(tf.random_normal([no_classes]))
```

变量的初始化可以是零，但初始化为随机正态分布可以提供稳定的训练。然后对输入进行加权并添加偏置以产生如下所示的 logits：

```
logits = tf.matmul(x_input, weights) + bias
```

感知机产生的 logits 必须与独热标签 y_input 进行比较。从第 1 章中得知，最好使用 softmax 和交叉熵来比较 logits 和独热标签。

TensorFlow 的 tf.nn.softmax_cross_entropy_with_logits API 做到了这一点。可以通过对交叉熵求平均值来计算损失。然后通过 tf.train.GradientDescentOptimizer 完成的梯度下降优化来得到交叉熵。优化器获取损失并将其最小化，这里的学习速率设为 0.5。下面显示了 softmax、交叉熵和损失的计算，以及随后的优化：

```
softmax_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
    labels=y_input, logits=logits)
loss_operation = tf.reduce_mean(softmax_cross_entropy)
optimiser = tf.train.GradientDescentOptimizer(
    learning_rate=0.5).minimize(loss_operation)
```

softmax 和交叉熵是靠 tf.nn 包一起计算的，该包中有其他几种有用的方法。tf.train

有几个优化器，我们在这里使用的是 vanilla 梯度下降。你可以访问 TensorFlow API 文档以了解其他可选参数。到目前为止，占位符、变量和操作都已定义，但尚未用在张量上。

> 请在 tensorflow.org 网站阅读 TensorFlow 中可用的优化器列表。Adam 优化器对计算机视觉应用程序特别有用。它通常会更快地收敛，并且不需要先定义学习速率。如要了解有关优化程序的理论总结，请访问 http://ruder.io/optimizing-gradient-descent。

3. 用数据训练模型

现在已经定义了模型和训练操作，下一步是开始用数据来训练模型。在训练过程中，计算梯度并更新权重。变量尚未初始化。接下来，启动会话并使用全局变量初始化器初始化变量。

```
session = tf.Session()
session.run(tf.global_variables_initializer())
```

本书中的大多数示例都需要上面的两行代码。默认读者将在需要的地方使用这两行代码。现在图已准备好接收数据并开始训练了。在循环中，批量读取数据并训练模型。训练模型通过运行有所需张量的会话来执行。必须调用优化器才能使图更新权重。

```
for batch_no in range(total_batches):
    mnist_batch = mnist_data.train.next_batch(batch_size)
    _, loss_value = session.run([optimiser, loss_operation], feed_dict={
        x_input: mnist_batch[0],
        y_input: mnist_batch[1]
    })
    print(loss_value)
```

`run` 方法的第一个参数可以是一个数组，用来接收输出值。这里传递损失是因为打印损失值可以告诉我们模型是否在被训练。因为目标是尽量减小损失，所以损失值预计将会减少。`feed dict` 是一个 Python 字典，它用于直接将输入和目标标签提供给占位符。一旦这个循环结束，损失值一般应该低于 0.46。接下来，可以通过计算准确率来评估模型的工作情况，如下所示：

```
predictions = tf.argmax(logits, 1)
correct_predictions = tf.equal(predictions, tf.argmax(y_input, 1))
accuracy_operation = tf.reduce_mean(tf.cast(correct_predictions,
                                            tf.float32))
test_images, test_labels = mnist_data.test.images, mnist_data.test.labels
accuracy_value = session.run(accuracy_operation, feed_dict={
    x_input: test_images,
    y_input: test_labels
})
print('Accuracy : ', accuracy_value)
session.close()
```

预测应该是最大激活的索引。应该将它与 MNIST 事实上的标签进行比较，以做出正确的预测。准确率是使用正确预测的平均值计算的。可以通过将测试数据作为传入字典来运行会话，以

评估数据的准确性。当整个程序运行时，它应该最终得到大约 90%的准确率。由于没有用更简单的 API 进行训练和测试，模型的定义可能过于明显。这个基本定义的级别提供了 TensorFlow 的表达能力。在接下来的几节中，我们将看到更高层次的 API。由感知机获得的准确率不是很高，下一节将使用一个更深的卷积网络层来提高准确性。

2.1.4 构建多层卷积网络

在本节中，我们将介绍如何在 TensorFlow 中创建多层卷积网络，并观察深层网络如何提高分类的准确率。我们将使用 TensorFlow 层的 API 来定义层，而不是从头做起。你可以体会这些方法中的最佳实践方法。和上一节一样导入库、数据集和占位符。这一次，我们将使用 TensorBoard 对训练过程进行可视化。为了可视化变量的统计，必须将变量统计的值添加到 `tf.summary` 中。

汇总（summary）将写入可被 TensorBoard 解释的文件夹。为了用 TensorBoard 来可视化，我们定义一个函数来写入汇总。

```
def add_variable_summary(tf_variable, summary_name):
  with tf.name_scope(summary_name + '_summary'):
    mean = tf.reduce_mean(tf_variable)
    tf.summary.scalar('Mean', mean)
    with tf.name_scope('standard_deviation'):
        standard_deviation = tf.sqrt(tf.reduce_mean(
            tf.square(tf_variable - mean)))
    tf.summary.scalar('StandardDeviation', standard_deviation)
    tf.summary.scalar('Maximum', tf.reduce_max(tf_variable))
    tf.summary.scalar('Minimum', tf.reduce_min(tf_variable))
    tf.summary.histogram('Histogram', tf_variable)
```

变量 `summary` 函数写入变量的汇总。这里有五个统计量添加到了汇总中：均值、标准差、最大值、最小值和直方图。汇总可以是标量或直方图。当记录多个变量时，我们将看到这些值如何在 TensorBoard 中可视化。与以前的模型不同，我们将 MNIST 数据调整为正方形，并像二维图像一样使用它。以下是将图像重整形为 28 像素 × 28 像素的命令：

```
x_input_reshape = tf.reshape(x_input, [-1, 28, 28, 1],
    name='input_reshape')
```

维度–1 表示批次大小可以是任何数。请注意，为了便于理解，有一个名为 `name` 的参数将会反映在 TensorBoard 图中。我们将定义一个二维卷积图层，其中定义了输入、滤波器、核和激活。该方法可以在任何地方被调用，以用于进一步的示例，并且在激活函数具有修正线性单元激活时是有用的。卷积函数层定义如下：

```
def convolution_layer(input_layer, filters, kernel_size=[3, 3],
                      activation=tf.nn.relu):
    layer = tf.layers.conv2d(
        inputs=input_layer,
        filters=filters,
```

```
            kernel_size=kernel_size,
            activation=activation,
        )
        add_variable_summary(layer, 'convolution')
        return layer
```

这里 kernel_size 和 activation 有默认参数。汇总被添加到函数内的层并返回相应的层。无论什么时候调用该函数，input_layer 都必须作为参数传递。这种定义将使代码简单小巧。以非常相似的方式，我们将为 pooling_layer 定义一个函数，如下所示：


```
def pooling_layer(input_layer, pool_size=[2, 2], strides=2):
    layer = tf.layers.max_pooling2d(
        inputs=input_layer,
        pool_size=pool_size,
        strides=strides
    )
    add_variable_summary(layer, 'pooling')
    return layer
```

该层具有默认参数：pool_size 为[2, 2]，步幅为 2。这些参数通常工作良好，但可以在必要时进行更改。该层也添加了汇总。接下来我们将定义一个密集层，如下所示：

```
def dense_layer(input_layer, units, activation=tf.nn.relu):
    layer = tf.layers.dense(
        inputs=input_layer,
        units=units,
        activation=activation
    )
    add_variable_summary(layer, 'dense')
    return layer
```

定义的密集层具有激活函数的默认参数，并且添加了变量汇总。pooling_layer 从卷积层获取特征图，并使用池化大小和步幅将它缩小为原大小的一半。所有这些层都作为图连接在一起。没有初始化任何值。可以添加另一个卷积层，将来自第一个卷积层的采样特征转换为更好的特征。池化后，为便于馈送到密集层，我们可以将激活值重整形为线性形式。

```
convolution_layer_1 = convolution_layer(x_input_reshape, 64)
pooling_layer_1 = pooling_layer(convolution_layer_1)
convolution_layer_2 = convolution_layer(pooling_layer_1, 128)
pooling_layer_2 = pooling_layer(convolution_layer_2)
flattened_pool = tf.reshape(pooling_layer_2, [-1, 5 * 5 * 128],
                            name='flattened_pool')
dense_layer_bottleneck = dense_layer(flattened_pool, 1024)
```

卷积层之间的唯一区别是滤波器的大小。在各层之间适当变化维度很重要。卷积核和步幅参数的选择是任意的，这些数字是根据经验选择的。定义了两个卷积层，然后可以接一个全连接层。密集层 API 可以采用任何单维度向量，并将其映射到任意数量的隐藏单元，在这个例子中为 1024。隐藏层随后是 ReLU 激活函数导致的非线性计算。变量汇总也被添加到这一层。接下来是有一定丢弃率的丢弃层。保持高丢弃值会阻止网络的学习。根据何时使用它，可以将训练模式设置为

True 和 False。训练时将它设置为 True（默认为 False）。我们将不得不在计算准确率时改变这一点。因此，把它作为一个布尔值，并在训练期间传入。

```
dropout_bool = tf.placeholder(tf.bool)
dropout_layer = tf.layers.dropout(
        inputs=dense_layer_bottleneck,
        rate=0.4,
        training=dropout_bool
    )
```

丢弃层再次被送入一个称为 logits 的密集层。logits 是有激活函数并导出分类数量的最后一层。这些激活函数会对一个特定的类别有尖锐的输出，这个类别就是目标类别。最多可以获得 10 个激活函数。

```
logits = dense_layer(dropout_layer, no_classes)
```

logits 输出与上一节中创建的模型非常相似。现在 logits 可以像以前一样通过 softmax 层，随后进行交叉熵计算。在这里，我们添加了一个作用域名称，以便在 TensorBoard 中获得更好的可视化效果。

```
with tf.name_scope('loss'):
    softmax_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
        labels=y_input, logits=logits)
    loss_operation = tf.reduce_mean(softmax_cross_entropy, name='loss')
    tf.summary.scalar('loss', loss_operation)
```

这个 loss 函数可以通过 tf.train API 的方法进行优化。在这里，我们将使用 Adamoptimiser。学习速率不需要定义，并且在大多数情况下运行良好。

```
with tf.name_scope('optimiser'):
    optimiser = tf.train.AdamOptimizer().minimize(loss_operation)
```

准确率按照之前的方法计算，但为了正确的预测率和准确率计算，添加了名称作用域。

```
with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        predictions = tf.argmax(logits, 1)
        correct_predictions = tf.equal(predictions,tf.argmax(y_input,1))
    with tf.name_scope('accuracy'):
        accuracy_operation = tf.reduce_mean(
            tf.cast(correct_predictions, tf.float32))
tf.summary.scalar('accuracy', accuracy_operation)
```

准确率的标量汇总也添加了进来。下一步是开始会话，并像上一节那样初始化变量。这里不再重复这些行。汇总必须合并，写入训练和测试汇总的文件必须定义。

```
merged_summary_operation = tf.summary.merge_all()
train_summary_writer = tf.summary.FileWriter('/tmp/train',session.graph)
test_summary_writer = tf.summary.FileWriter('/tmp/test')
```

请注意，该图是使用 summary_writer 写入的。除了训练时的准确率计算和将这些值添加到汇总之外，训练与以前非常相似。接下来，可以批量加载数据并开始训练。

```
test_images, test_labels = mnist_data.test.images,mnist_data.test.labels

for batch_no in range(total_batches):
    mnist_batch = mnist_data.train.next_batch(batch_size)
    train_images, train_labels = mnist_batch[0], mnist_batch[1]
    _, merged_summary = session.run([optimiser, merged_summary_operation],
                                    feed_dict={
        x_input: train_images,
        y_input: train_labels,
        dropout_bool: True
    })
    train_summary_writer.add_summary(merged_summary, batch_no)
    if batch_no % 10 == 0:
        merged_summary, _ = session.run([merged_summary_operation,
                                        accuracy_operation],feed_dict={
            x_input: test_images,
            y_input: test_labels,
            dropout_bool: False
        })
        test_summary_writer.add_summary(merged_summary, batch_no)
```

每次迭代都会返回汇总数据，并将其添加到 writer 中。对于每个第十次迭代，都会添加测试汇总。请注意，丢弃仅在训练期间启用，而不在测试期间启用。我们已经完成了网络的定义和汇总，并可以运行它。要查看训练过程，可以按照第 1 章所述进入 TensorBoard。

利用 TensorBoard 深度学习

在浏览器中打开 TensorBoard 后，转到 Graphs 选项卡。我们已经定义并且正在训练的图应该会显示出来。右键单击节点，可以选择要从主图中删除的操作。经过一定程度的对准之后，应如图 2-2 所示。

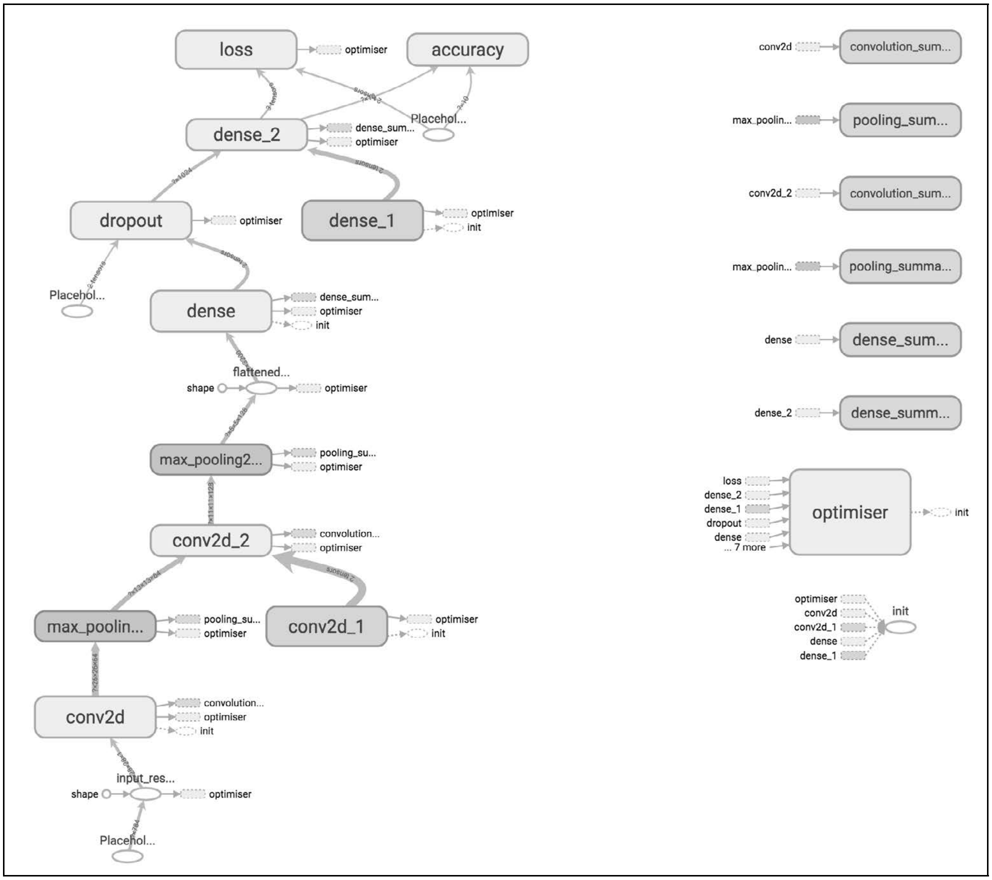


图　2-2

我们已定义的所有层都显示出来了。这对检查架构的定义非常有用。计算图的方向和所有细节都进行了很好的可视化。点击每个节点，可以看到节点的详细信息，如输入和输出张量形状，如图 2-3 所示。

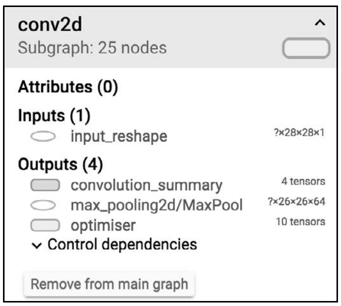


图　2-3

这些值可用于交叉检查各层参数的定义。查看左下角的图例，熟悉一下这个页面，如图 2-4 所示。

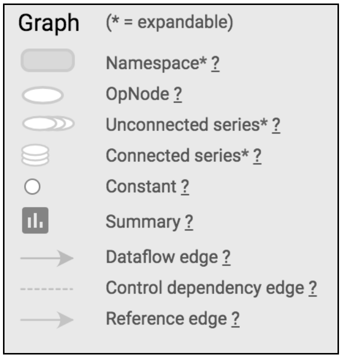


图 2-4

名称作用域被分组，单击节点上的加号可以看到各个组件。节点按颜色排列。现在我们可以进入标量页面。通过在页面中移动，你可以看到准确率图，如图 2-5 所示。

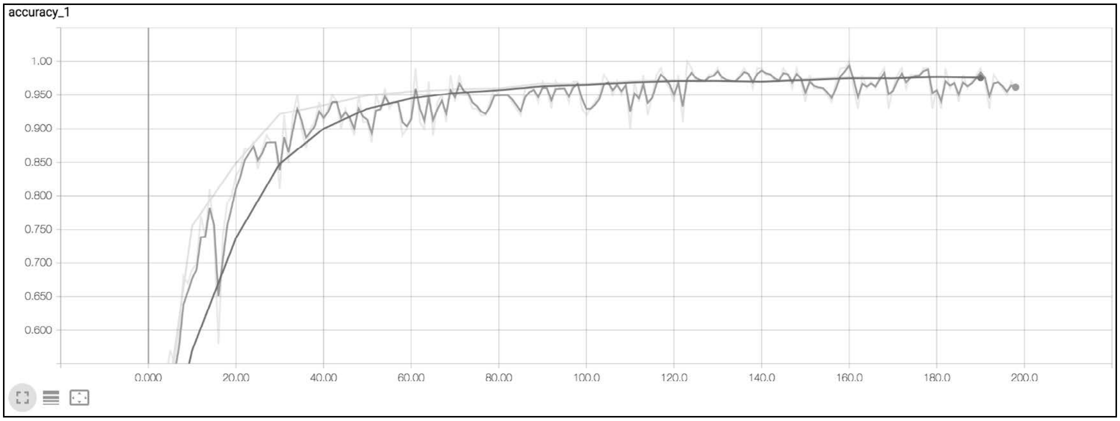


图 2-5

橙色线表示训练数据，蓝色线表示测试数据。[①]它们大致遵循相同的模式。浅色线表示原始值，深色线则是平滑过的曲线。平滑系数可以在 UI 中选择。测试数据的准确率达到了 97%以上。图 2-6 是损失汇总中的图。

① 本书彩图可到图灵社区本书主页（http://ituring.cn/book/2396）“随书下载”处查看。——编者注

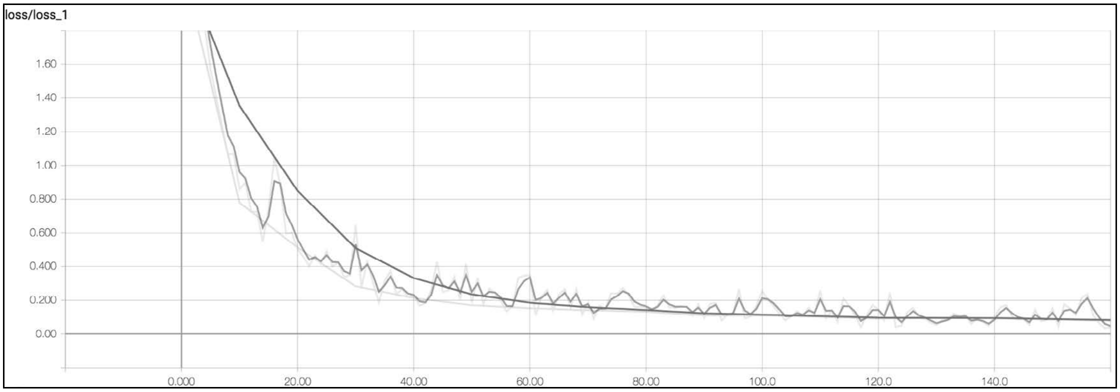


图　2-6

在训练过程中，训练数据和测试数据的损失稳步下降，这是一个好迹象。所有汇总数据将在训练过程中进行更新，我们可以看到准确性的提高以及损失的减少，从而获得 97.38%的测试准确率。

这有助于你了解该模型是否正在学习并朝着更好的结果前进。其他汇总，如最小值、最大值、平均值和标准偏差，也有用。图 2-7 是密集层的图。

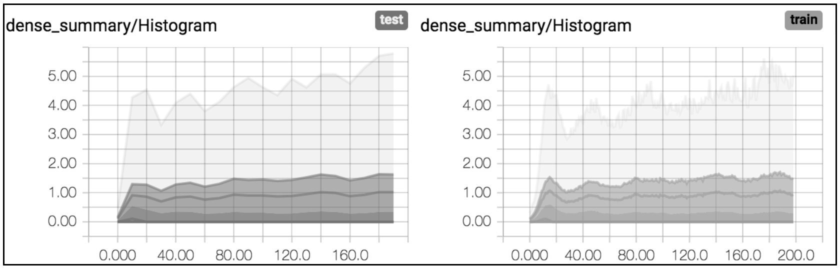


图　2-7

这些汇总对于注意权重的变化很有用。这些分布也可以可视化为直方图，如图 2-8 所示。

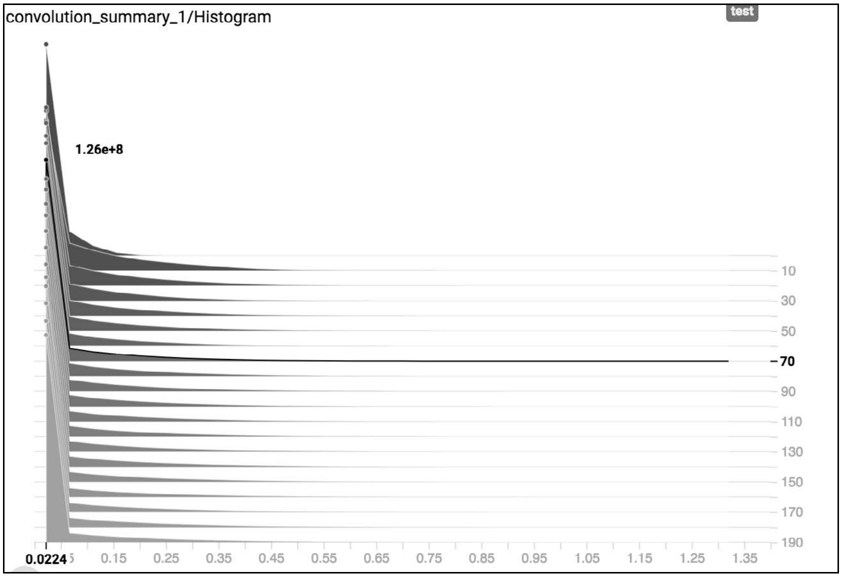


图　2-8

以上是 `logits` 权重的散布排列。这些是 TensorBoard 提供的美观的可视化，对训练非常有帮助。通过更深的模型，我们能够看到准确率的显著提高。在下一节中，我们将了解如何使用 Keras API 来训练相同的模型。现在你可以看到 TensorBoard 在检查深度学习模型和训练过程中的强大功能。

2.2　在 Keras 中训练 MNIST 模型

在本节中，我们将使用 `tf.keras` API 建立与上一节相同的模型。最好从 TensorFlow 中学习 Keras 和各层的包，因为在几个开源代码中可以看到它们。本书的目标是让你了解 TensorFlow 的各种功能，以便在上面构建产品。

> “阅读代码的次数比编写代码的次数多。”

考虑到前面这句引语，你将会看到如何使用各种 API 实现相同的模型。实现最新算法的开源代码将是这些 API 的混合。接下来，让我们从 Keras 实现开始。

2.2.1　准备数据集

MNIST 数据与 Keras 一起提供。首先，导入 TensorFlow。然后定义一些常量，例如批次规模、类别和周期（epoch）数。可以根据你的计算机上可用的 RAM 大小选择批次规模。批次越大，所需的 RAM 就越多。批次规模对准确率的影响很小。分类的数量在这个例子中等于 10，当然，不

同的问题会有所不同。周期的数量决定了训练必须经过整个数据集的次数。如果在所有周期结束时损失减少，则可以将其设置为一个较高的数字。在一些情况下，长时间训练可以提高准确率。让我们来看看创建数据集的步骤。

(1) 设置输入图像的尺寸。

```
batch_size = 128
no_classes = 10
epochs = 2
image_height, image_width = 28, 28
```

(2) 使用 Keras 工具将数据加载到内存中。

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

(3) 将向量重整形为图像格式，并使用给出的代码定义卷积的输入维度。

```
x_train = x_train.reshape(x_train.shape[0], image_height,image_width,1)
x_test = x_test.reshape(x_test.shape[0], image_height, image_width, 1)
input_shape = (image_height, image_width, 1)
```

(4) 将数据类型转换为浮点数。

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

(5) 减去数据的平均值以归一化数据。

```
x_train /= 255
x_test /= 255
```

(6) 将分类标签转换为独热编码。

```
y_train = tf.keras.utils.to_categorical(y_train, no_classes)
y_test = tf.keras.utils.to_categorical(y_test, no_classes)
```

这与 TensorFlow 的代码编写方式有很大不同。数据已经加载到内存中，这里没有任何占位符的概念。

2.2.2　构建模型

在本节中，我们将使用一些卷积层，然后使用全连接层来训练前面的数据集。建立一个简单的序列模型，其中包含两个卷积层，然后是池化层、丢弃层和密集层。一个序列模型具有将层堆叠在另一层上的 `add` 方法。第一层有 64 个滤波器，第二层有 128 个滤波器。所有滤波器的核大小均为 3。在卷积层之后应用最大池化。卷积层的输出被平坦化，连接到具有丢弃层连接的几个全连接层。

最后一层连接到 softmax，因为这是一个多分类问题。以下代码展示了如何定义模型。

```
def simple_cnn(input_shape):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.Conv2D(
        filters=64,
        kernel_size=(3, 3),
        activation='relu',
        input_shape=input_shape
    ))
    model.add(tf.keras.layers.Conv2D(
        filters=128,
        kernel_size=(3, 3),
        activation='relu'
    ))
    model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
    model.add(tf.keras.layers.Dropout(rate=0.3))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Dense(units=1024, activation='relu'))
    model.add(tf.keras.layers.Dropout(rate=0.3))
    model.add(tf.keras.layers.Dense(units=no_classes,
activation='softmax'))
    model.compile(loss=tf.keras.losses.categorical_crossentropy,
                  optimizer=tf.keras.optimizers.Adam(),
                  metrics=['accuracy'])
    return model
simple_cnn_model = simple_cnn(input_shape)
```

该模型刚定义好，必须进行编译。在编译时，必须定义损失、优化器和度量指标。损失是交叉熵，优化通过 Adam 算法进行，用准确率作为度量指标。使用加载的数据来训练和评估数据。使用训练参数加载训练数据并拟合模型。

```
simple_cnn_model.fit(x_train, y_train, batch_size, epochs, (x_test, y_test))
train_loss, train_accuracy = simple_cnn_model.evaluate(
    x_train, y_train, verbose=0)
print('Train data loss:', train_loss)
print('Train data accuracy:', train_accuracy)
```

使用 Keras API 时不会创建会话。如下评估测试数据：

```
test_loss, test_accuracy = simple_cnn_model.evaluate(
    x_test, y_test, verbose=0)
print('Test data loss:', test_loss)
print('Test data accuracy:', test_accuracy)
```

评估也是在没有显式创建会话的情况下进行的。运行结束后，结果应该如下所示：

```
Loss for train data: 0.0171295607952
Accuracy of train data: 0.995016666667
Loss for test data: 0.0282736890309
Accuracy of test data: 0.9902
```

在测试数据上得到了高于 99%的准确率。训练数据达到的准确率高于测试数据。把它们都打印出来始终是一种很好的做法。准确率的差异是由于迭代的次数不同。由于数据集的差异，准确率比之前在 TensorFlow 中创建的模型稍微高一些。

2.3　其他流行的图像测试数据集

MNIST 数据集是测试算法最常用的数据集。但还有其他用于测试图像分类算法的数据集。

2.3.1　CIFAR 数据集

CIFAR-10（canadian institute for advanced research-10）数据集有 60 000 幅图像，其中 50 000 幅用于训练，10 000 幅用于测试。分类的数量是 10，图像尺寸是 32 像素 × 32 像素。图 2-9 是从每个类别中随机选择的图像。



图　2-9

这些图像很小，只包含一个对象。CIFAR-100 数据集包含相同数量的图像，但有 100 个类别。因此，每个类别只有 600 幅图像。每幅图像都带有粗标签和精标签。如果你想实验，这个数据集可以在 `tf.keras.datasets` 上找到。

2.3.2　Fashion-MNIST 数据集

Fashion-MNIST 是作为 MNIST 数据集的备选项创建的。这个数据集被认为太简单了，可以直接用 MNIST 替换。

图 2-10 是执行**主成分分析**（PCA）后从数据集中随机选择的示例。

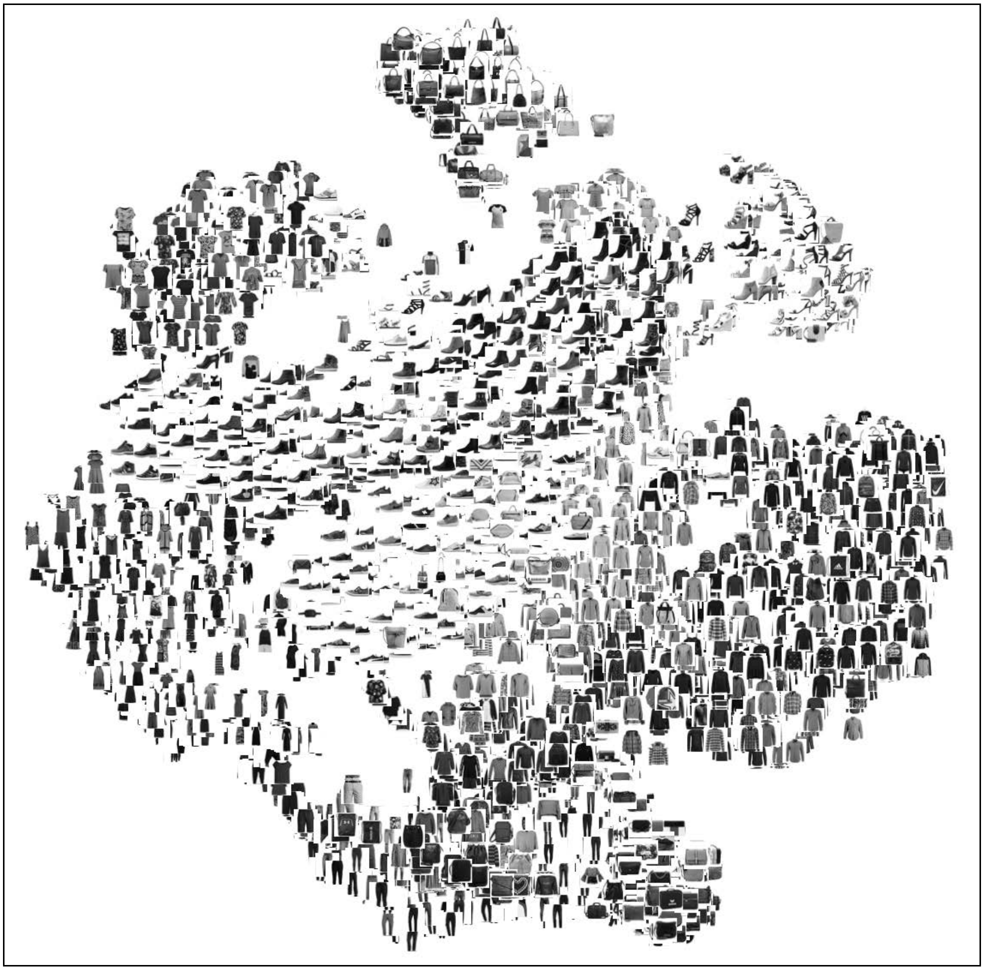


图　2-10

数据集规模、标签数量和图像大小与 MNIST 类似。更多细节可以在 https://github.com/zalandoresearch/fashion-mnist 找到。你可以运行先前学习的模型并检查准确率。

2.3.3　ImageNet 数据集和竞赛

ImageNet 是一个计算机视觉数据集，包含 14 197 122 幅图像，21 841 个 Synset 索引。Synset 是 WordNet 层次结构中的一个节点，又是一组同义词集合。每年都有基于该数据集的 1000 个类别的比赛。它一直是评估图像分类算法性能的基准。

2013 年，一个基于深度学习的计算机视觉模型获得了第一名。从那时起，只有深度学习模型赢得了比赛。图 2-11 是比赛多年来排名前五的错误率。

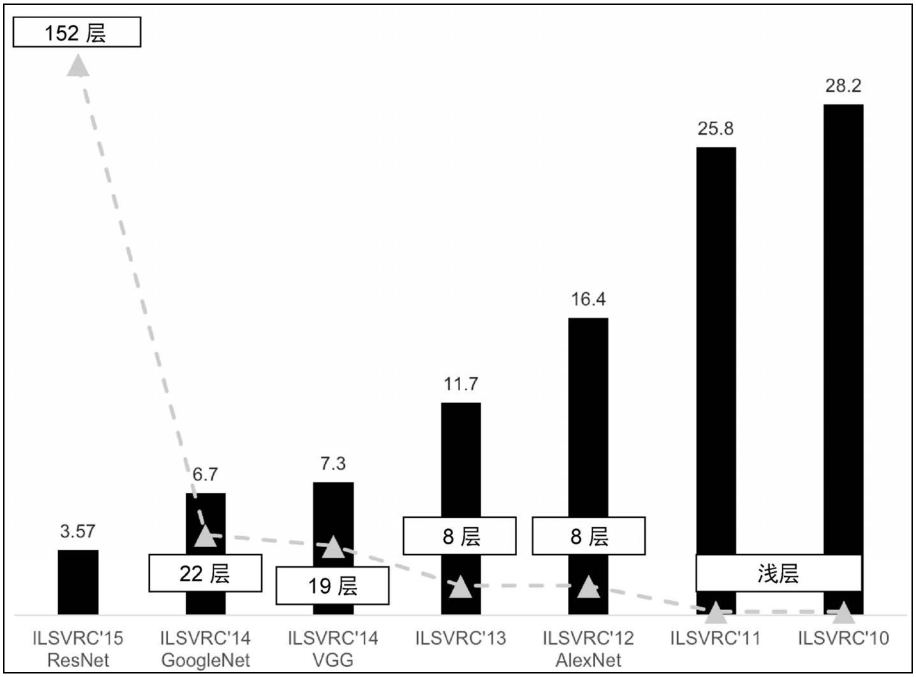


图　2-11

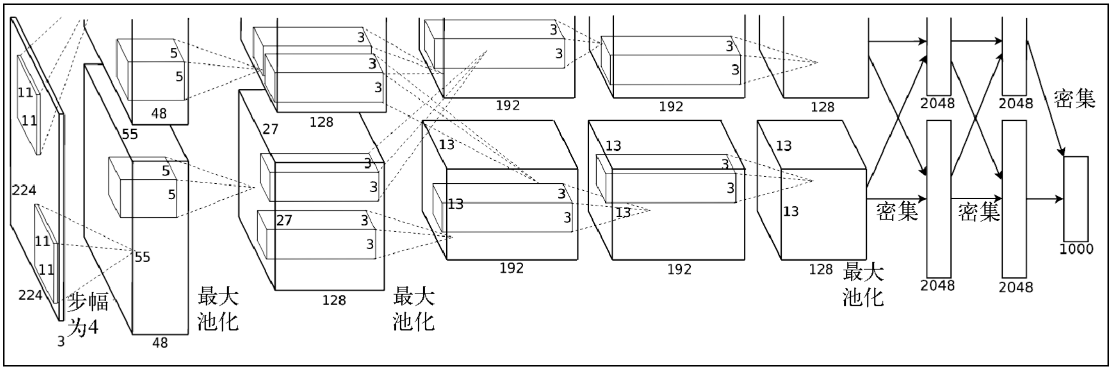
可以注意到多年来准确率以及图的深度一直在提升。接下来将介绍该图中展示的模型。

2.4　更大的深度学习模型

我们将介绍几个在 ImageNet 竞赛中取得最新成果的模型。

2.4.1　AlexNet 模型

AlexNet 是第一个激起人们对计算机视觉深度学习浓厚兴趣的发表成果。Krizhevsky 等人提出了 AlexNet。该模型是该领域的先驱并很有影响力。该模型在 2013 年赢得了 ImageNet 竞赛，错误率只有 15.4%，明显好于第二名。该模型相对简单，具有五个卷积层。它面临的挑战是对 1000 个类别的对象进行分类。图像和数据有 1500 万幅带标注的图像和超过 22 000 个类别。其中，只有 1000 个类别用于比赛。AlexNet 使用 `ReLU` 作为激活函数，并发现它的训练速度比其他激活函数快几倍。图 2-12 显示的是模型的体系结构。



* 经 Krizhevsky 等人许可转载

图　2-12

该论文还使用了数据增强技术，如图像翻译、水平翻转和随机裁剪。丢弃层可防止过拟合。该模型使用 vanilla SGD 进行训练。SGD 的参数经过仔细选择用于训练。在一组固定集合的训练迭代中变化学习速率。动量和权重衰减需要固定的训练值。该论文介绍了一种称为**本地响应归一化**（local response normalization，LRN）的概念。LRN 层对若干滤波器中的每个像素进行归一化处理，以避免某个滤波器中的巨大激活。

最近的研究表明，由于 LRN 并没有带来太大的性能改善，因此这一层已经不再使用了。AlexNet 总共有 6000 万个参数。

2.4.2　VGG-16 模型

VGG 模型代表牛津大学的 Visual Geometry Group。该模型非常简单，并且比 AlexNet 具有更大的深度。这篇论文有两个模型，分别有 16 层和 19 层。所有的 CNN 层都使用步幅为 1、填充大小为 2 的 3 × 3 滤波器，以及一个规模为 2、步幅为 2 的最大池化。这导致参数数量的减少。尽管由于最大池化，模型的规模正在减小，但滤波器数量随着层数的增加而增加。16 层的深度模型的架构如图 2-13 所示。

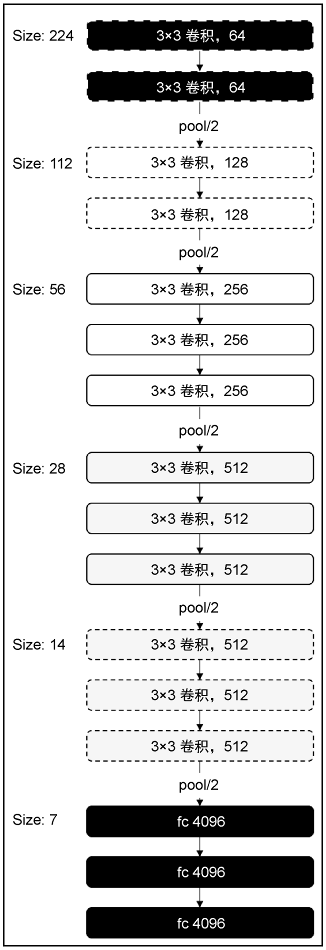


图　2-13

该模型拥有 1.38 亿个参数，是这些模型中最大的。但参数的均匀性相当好。其特点是，网络越深，随着滤波器数量的增多，图像越小。其中一项数据增强技术是尺度抖动（scale jittering）。在该增强技术中，具有随机大小的一个边被认为是可以改变尺度的。

2.4.3　谷歌 Inception-V3 模型

Inception-V3 由 Szegedy 等人提出，并引入了具有更好泛化方式的 inception 概念。该模型在 2014 年赢得了 ImageNet 竞赛。它的速度和规模带来了很高的效率。它的参数是 AlexNet 的十二分之一。inception 是构建宏架构的微架构。每个隐藏层都具有图像的更高级别表示。每一层都有使用池化或其他层的选项。inception 使用几个核，而不是只使用一种类型的核。平均池化之后是不同大小的卷积，然后将它们级联起来。

核参数可以根据数据进行学习。由于使用几个核，该模型可以检测小的特征以及更高的抽象。1×1卷积将减少特征并因此减少计算量，在推断期间占用更少的RAM。图2-14展示了最简单形式的inception模块，其中有各种核大小和池化的卷积选项。

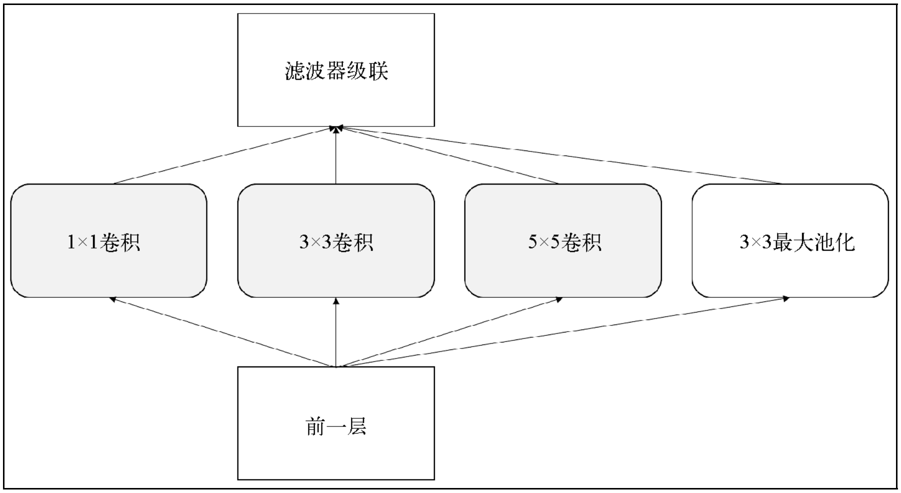


图　2-14

注意，不同于AlexNet和VGG，这里的操作是并行发生的。输出的容积量（volume）很大，因此引入了1×1滤波器以降低维度。当降低的维度被添加到体系架构时，如图2-15所示。

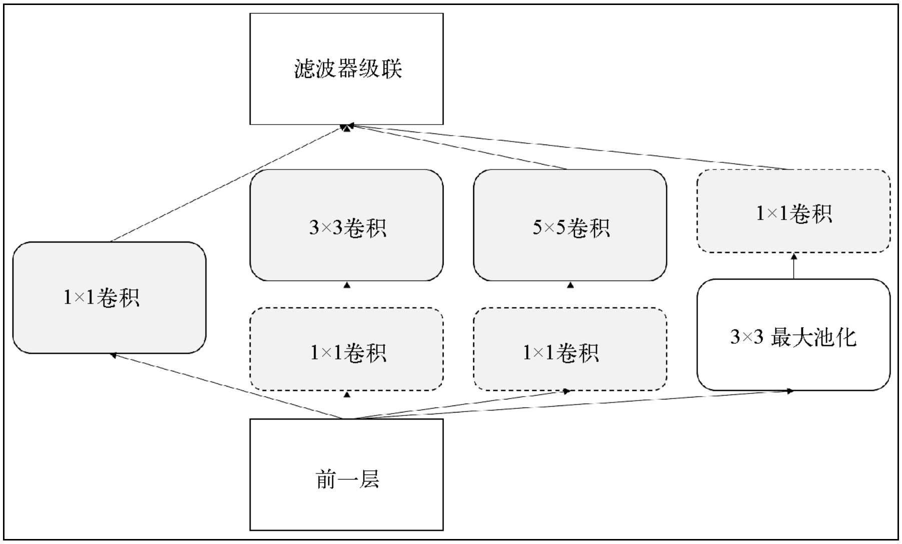
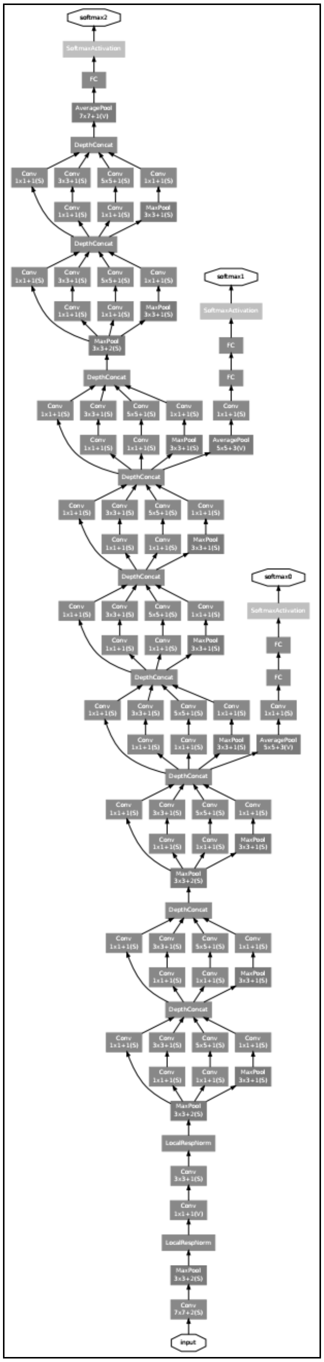


图　2-15

这个模型的整个架构如图2-16所示。



* 经 Szegedy 等人许可转载

图　2-16

该模型有 9 个 inception 模块，共 100 层，并且它们实现了良好的性能。

2.4.4　微软 ResNet-50 模型

ResNet 由 He 等人提出，并在 2015 年赢得了 ImageNet 竞赛。这种方法表明可以训练更深层的网络。网络越深，准确率就越趋近于饱和。这甚至不是因为过拟合或存在大量参数，而是因为减小了训练误差。这是由无法反向传播梯度导致的。可以通过将梯度直接发送到具有残差块的较深层来克服，如图 2-17 所示。

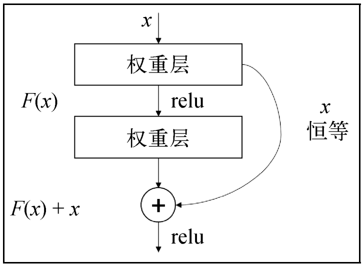


图　2-17

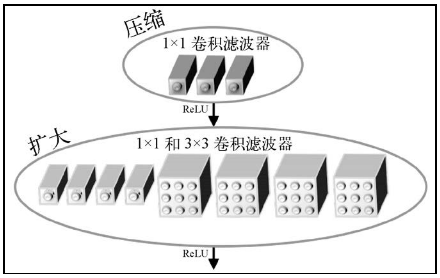
每两层连接形成一个残差块。你可以看到训练是在各层之间传递的。通过这种技术，反向传播可以将误差带到更早的层。

模型定义可参考 https://github.com/tensorflow/tensorflow/tree/r1.4/tensorflow/python/keras/_impl/keras/applications。模型的每一层都定义了，并且也可以找到在 ImageNet 数据集上预训练的权重。

2.4.5　SqueezeNet 模型

为缩小模型规模和减少参数数量，Iandola 等人引入了 SqueezeNet 模型。

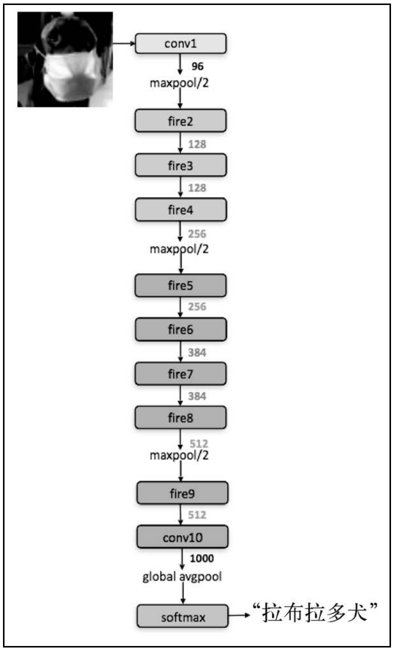
通过用 1 × 1 滤波器替换 3 × 3 滤波器，网络变得更小，如图 2-18 所示。



* 经 Iandola 等人许可转载

图　2-18

在更高级别发生时，3 × 3 滤波器的输入数量也减少了层的下采样，提供了大的激活图（activation map），如图 2-19 所示。

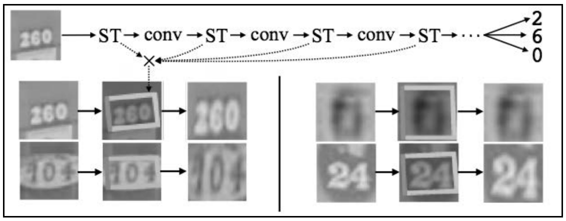


* 经 Iandola 等人许可转载

图　2-19

2.4.6　空间变换网络模型

Jaderberg 等人提出了空间变换网络（spatial transformer network）模型，该模型尝试在传递给 CNN 之前转换图像。这与其他网络不同，因为它试图在卷积之前修改图像。该网络通过学习参数来转换图像。参数的学习是通过仿射变换（affine transformation）进行的。使用仿射变换，可以实现空间不变性。在以前的网络中，空间不变性是通过最大池化层实现的。空间变换网络的布局如图 2-20 所示。



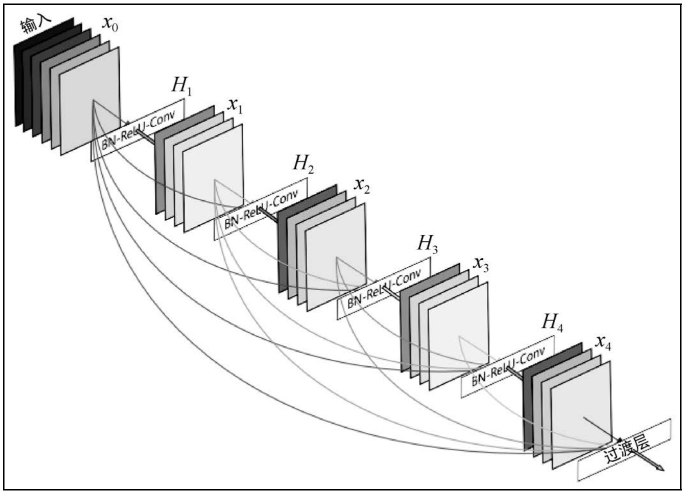
* 经 Jaderberg 等人许可转载

图　2-20

2.4.7　DenseNet 模型

DenseNet 是 Huang 等人提出的 ResNet 的扩展。在 ResNet 的块中，前一层通过求和合并到未来层。在 DenseNet 中，前一层通过级联合并到未来层。DenseNet 将所有层连接到以前的层，并将当前层连接到下一层。

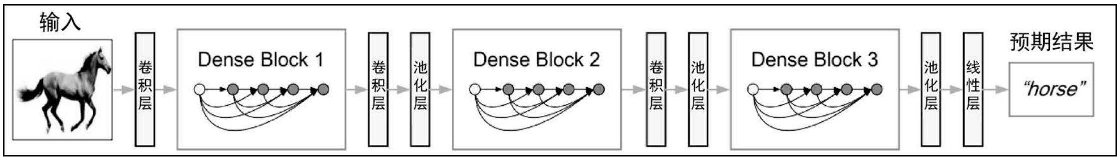
在图 2-21 中，可以看到如何将特征图作为输入提供给其他层。



* 经 Huang 等人许可转载

图　2-21

它提供了几个优点，如平滑梯度、特征变换等。它也减少了参数的数量，如图 2-22 所示。



* 经 Huang 等人许可转载

图　2-22

我们已经介绍了图像分类任务的所有最新算法。任何一个架构都可以用于图像分类任务。下一节将介绍如何使用这些高级架构训练模型以预测宠物，并提高准确率。

2.5　训练猫与狗的模型

在本节中，我们将准备并训练一个预测猫与狗的模型，并了解一些提高准确率的技巧。大部分的图像分类问题都属于这种范式。本节介绍的技术（如数据增强和迁移学习）对于一些问题很有用。

2.5.1 准备数据

为了进行分类，我们从 kaggle 下载数据并以适当的格式进行存储。注册并登录 kaggle 网站，转至 https://www.kaggle.com/c/dogs-vs-cats/data。从该页面下载 train.zip 和 test1.zip 文件。train.zip 文件包含 25 000 个宠物图像的数据。我们只使用一部分数据来训练一个模型。具有更多计算能力（GPU）的读者可以使用比建议更多的数据。运行以下脚本重新排列图像并创建必要的文件夹：

```
import os
import shutil

work_dir = '' # give your correct directory
image_names = sorted(os.listdir(os.path.join(work_dir, 'train')))

def copy_files(prefix_str, range_start, range_end, target_dir):
    image_paths = [os.path.join(work_dir, 'train', prefix_str + '.' +
str(i) + '.jpg')
                   for i in range(range_start, range_end)]
    dest_dir = os.path.join(work_dir, 'data', target_dir, prefix_str)
    os.makedirs(dest_dir)
    for image_path in image_paths:
        shutil.copy(image_path, dest_dir)

copy_files('dog', 0, 1000, 'train')
copy_files('cat', 0, 1000, 'train')
copy_files('dog', 1000, 1400, 'test')
copy_files('cat', 1000, 1400, 'test')
```

对于该实验，我们只使用 1000 幅猫和狗的图像。因此，将图像 0 ~ 999 从下载的文件夹复制到 cats 下新创建的 train 文件夹。同样，将 1000 ~ 1400 复制到 data/test/cat 中，将 10 ~ 999 复制到 train/dogs 中，将 1000 ~ 1400 复制到 data/test/ dog 中，这样就可以为每个类别复制 1000 个训练样本，并为每个类别复制 400 个验证示例。

2.5.2 使用简单 CNN 进行基准测试

让我们在这个数据集上运行以前的 `simple_cnn` 模型，看看它是如何执行的。这个模型的性能将成为我们判断其他技术的基准。我们将为数据加载和训练操作定义一些变量，如下所示：

```
image_height, image_width = 150, 150
train_dir = os.path.join(work_dir, 'train')
test_dir = os.path.join(work_dir, 'test')
no_classes = 2
no_validation = 800
epochs = 2
batch_size = 200
no_train = 2000
no_test = 800
input_shape = (image_height, image_width, 3)
epoch_steps = no_train // batch_size
test_steps = no_test // batch_size
```

这个常数用于本节讨论的预测猫与狗的训练模型的技术。这里，我们使用 2800 幅图像来训练和测试，这个数量对于个人计算机的 RAM 是合理的。但对于更大的数据集来说，这是不可持续的。如果我们一次只加载一批图像进行训练和测试会更好。为此，`tf.keras` 有一个名为 `ImageDataGenerator` 的类，可以只在必要时读取图像。假设从前面的部分导入了一个 `simple_cnn` 模型。以下是使用生成器加载图像的示例：


```
generator_train =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. / 255)
generator_test =
tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. / 255)
```

这个定义在加载时也会重新缩放图像。接下来，我们可以使用 `flow_from_directory` 方法从目录中读取图像，如下所示：

```
train_images = generator_train.flow_from_directory(
    train_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height))

test_images = generator_test.flow_from_directory(
    test_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height))
```

加载图像的目录、图像的批次规模和目标大小都作为参数传递。该方法执行重新缩放并分批传递数据以拟合模型。该生成器可以直接用于拟合模型。可以像下面这样使用模型的 `fit_generator` 方法：

```
simple_cnn_model.fit_generator(
    train_images,
    steps_per_epoch=epoch_steps,
    epochs=epochs,
    validation_data=test_images,
    validation_steps=test_steps)
```

该模型拟合来自训练图像生成器的数据。周期的数量由训练来定义，传递验证数据以获得模型过度训练的性能。这个 `fit_generator` 可以并行处理数据和模型训练。CPU 执行缩放，而 GPU 可以执行模型训练。这样可带来计算资源的高效率。经过 50 个周期后，这个模型应该有 60% 的准确率。接下来看看如何通过增强数据集来改进性能。

2.5.3 增强数据集

数据增强提供了扩大数据集规模的方法。数据增强在训练期间引入噪声，从而在模型中产生对各种输入的稳健性。该技术在数据集很小并且可以与其他技术结合使用的场景中非常有用。接下来介绍不同类型的增强。

增强技术

增强图像的各种方式如下。

- 翻转：镜像或水平/垂直方向翻转图像。
- 随机裁剪：随机的部分被裁剪，因此该模型可以处理遮挡。
- 剪切：图像变形以影响对象的形状。
- 缩放：图像的缩放部分被训练以处理不同比例的图像。
- 旋转：旋转对象以处理对象中各种程度的变化。
- 白化：通过主成分分析完成，只保留重要数据。
- 归一化：通过标准化均值和方差来规范像素。
- 通道移动：移动颜色通道，使模型对各种伪影引起的颜色变化更稳健。

所有这些技术都是在 `ImageDataGenerator` 中实现的，以增加数据集的大小。以下是 `generator_train` 的一个修改版本，其中包含前面讨论的一些增强技术：

```
generator_train = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1. / 255,
    horizontal_flip=True,
    zoom_range=0.3,
    shear_range=0.3,)
```

在前面的代码中替换 `generator_train` 会将准确率提高到 90%。更改增强的参数并注意这些更改。我们将在下一节讨论一种称为迁移学习的技术，这有助于以更少的数据训练更大的模型。

2.5.4　迁移学习或微调模型

迁移学习是从一个大型数据集上预训练过的模型中学习的过程。用随机初始化训练一个模型通常需要时间和能量来获得结果。使用预训练的模型来初始化模型可以更快地收敛，节省时间和精力。这些预训练的模型通常经过了精心挑选的超参数训练。

预训练模型的一些层可以不经任何修改而使用，或者进行一些训练以适应变化。在本节中，我们将学习如何对在数百万个类别的 ImageNet 数据集上训练过的模型进行微调或迁移学习。

训练瓶颈特征

前面介绍的模型很简单，因此准确率可能会较低。应该以这些简单的模型为基础建立复杂的模型，而不能从头开始构建。因此，需要提取瓶颈特征并对分类器进行训练。瓶颈特征是由复杂的架构训练数百万幅图像产生的特征。通过前向传递完成图像训练，并且最终层的前一层的特征被存储下来。然后，用一个简单的逻辑分类器进行分类训练。瓶颈层提取按如下方式进行：

```
generator = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1. / 255)

model = tf.keras.applications.VGG16(include_top=False)
```

```
train_images = generator.flow_from_directory(
    train_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height),
    class_mode=None,
    shuffle=False
)
train_bottleneck_features = model.predict_generator(train_images, epoch_steps)

test_images = generator.flow_from_directory(
    test_dir,
    batch_size=batch_size,
    target_size=(image_width, image_height),
    class_mode=None,
    shuffle=False
)

test_bottleneck_features = model.predict_generator(test_images, test_steps)
```

VGG 模型被用于预测图像。标签分配如下：

```
train_labels = np.array([0] * int(no_train / 2) + [1] * int(no_train / 2))
test_labels = np.array([0] * int(no_test / 2) + [1] * int(no_test / 2))
```

具有几个层的序列模型被构建、编译并用瓶颈特征训练。可以用如下代码来实现：

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=train_bottleneck_features.shape[1:]))
model.add(tf.keras.layers.Dense(1024, activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(1, activation='softmax'))
model.compile(loss=tf.keras.losses.categorical_crossentropy,
              optimizer=tf.keras.optimizers.Adam(),
              metrics=['accuracy'])
```

使用如下代码通过模型训练这些瓶颈特征：

```
model.fit(
    train_bottleneck_features,
    train_labels,
    batch_size=batch_size,
    epochs=epochs,
    validation_data=(test_bottleneck_features, test_labels))
```

这给训练模型提供了一种不同的方法，并且在训练数据较少时很有用。这通常是训练模型更快的方法。仅使用预训练模型中最后的激活函数来适应新任务。这个想法可以扩展到微调一些层。

2.5.5 在深度学习中微调一些层

可以加载一个预训练的模型，并仅训练几个层。当给定的问题与模型训练的图像非常不同时，这种方法的效果更好。微调是深度学习中常见的做法。这在数据集较小时具有优势，优化也可以更快地获得。

在小数据集上训练深度网络会导致过拟合。使用微调过程可以避免这种过拟合。在较大数据集上训练的模型也应该相似，因为我们希望激活函数和特征与较小数据集类似。你可以从存储的权重路径开始，如下所示：

```
top_model_weights_path = 'fc_model.h5'
```

加载 VGG 模型并将初始层设置为不可训练。VGG 模型将在后面的章节详细介绍。现在，将 VGG 视为一种在图像数据上运行良好的深度学习模型。使用以下代码用新的可训练层替换全连接层：

```
model = tf.keras.applications.VGG16(include_top=False)
```

可以在 VGG 模型之上建立一个小型的双层前馈网络，通常使用隐藏单元、激活和丢弃，如下所示：

```
model_fine_tune = tf.keras.models.Sequential()
model_fine_tune.add(tf.keras.layers.Flatten(input_shape=model.output_shape))
model_fine_tune.add(tf.keras.layers.Dense(256, activation='relu'))
model_fine_tune.add(tf.keras.layers.Dropout(0.5))
model_fine_tune.add(tf.keras.layers.Dense(no_classes, activation='softmax'))
```

顶层模型还需要加载已经完全训练过的权重。然后可以将顶层模型添加到卷积基中：

```
model_fine_tune.load_weights(top_model_weights_path)
model.add(model_fine_tune)
```

我们可以将前 25 层直到最后一个卷积块设置为不可训练，这样它们的权重就不会被更新。只有其他层会被更新。

```
for vgg_layer in model.layers[:25]:
    vgg_layer.trainable = False
```

使用梯度下降优化器以量级为 4 的低学习速率编译模型：

```
model.compile(loss='binary_crossentropy',
              optimizer=tf.keras.optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])
```

我们可以将之前介绍的增强技术与剪切、缩放和翻转相结合。生成器可以添加来自目录的流、训练数据集和验证数据集。现在该模型可以与数据增强结合进行微调。这种训练方式比以前的所有方法都有更好的准确性。表 2-1 是使用迁移学习的指导。

表　2-1

数据规模	相似数据集	不同数据集
小数据	微调输出层	微调更深的层
大数据	微调整个模型	从头训练

根据数据大小，可以确定要微调的层数。数据越少，要微调的层数就越少。我们已经看到如何使用迁移学习技术来提高模型的准确率。

2.6　开发现实世界的应用

识别出猫和狗是一个很酷的问题，但不太可能成为一个重要问题。实际产品中使用的图像分类可能不同。你可能有不同的数据、目标等。在本节中，你将学习解决这些不同设置的技巧和窍门。解决新问题时要考虑的因素如下。

- 目标数量：这是一个 10 个类别的问题还是 10 000 个类别的分类问题？
- 类内差异有多大？例如，不同类型的猫是否必须在一个类别标签下进行识别？
- 类间差异有多大？例如，是否需要识别不同的猫？
- 数据规模有多大？
- 数据的平衡程度如何？
- 是否已经有一个用很多图像训练的模型？
- 部署推断时间和模型规模的必要条件是什么？是在 iPhone 上 50 毫秒还是在 Google 云平台上 10 毫秒？可以消耗多少内存来存储模型？

在处理图像分类问题时，尝试回答这些问题。基于这些答案，我们可以设计训练架构并提高准确率，如下一节所述。

2.6.1　选择正确的模型

架构有很多选项。基于部署的灵活性，你可以选择模型。请记住，卷积层更小、更慢，密集层则更大、更快。在规模、运行时间和准确率上要进行权衡。在最终决定之前测试所有架构是明智之举。基于应用场景，某些模型可能比其他模型更好。你可以减小输入规模以加快推测速度。可以根据下节所述的指标来选择架构。

2.6.2　处理欠拟合和过拟合场景

一个模型对于解决的问题有时可能太小或太大。这可以分别被归类为**欠拟合**和**过拟合**。当模型太小时会出现欠拟合，当训练准确率较低时可以进行测量。当模型太大并且训练准确率和测试准确率之间存在很大差距时会出现过拟合。

欠拟合可以通过以下方法解决：

- 获取更多数据；
- 尝试一个更大的模型；
- 如果数据很小，尝试迁移学习技术或进行数据增强。

过拟合可以通过以下方法解决：

- 使用丢弃和批归一化等技术进行正则化；

- 增强数据集。

时刻注意损失值的变化。迭代过程中损失应该减少，如果损失不减少，则表示训练已停止。一个解决方案是尝试一个不同的优化器。类别失衡可以通过加权损失函数来处理。总是使用 TensorBoard 查看汇总。很难估计需要多少数据。本节是训练任何深度学习模型的最佳课程。接下来将介绍一些特定于应用的指导原则。

2.6.3　从面部检测性别和年龄

一些应用可能需要从面部检测性别和年龄。人脸图像可以通过人脸检测器获得。裁剪后的人脸图像可以作为训练数据，并且应该给出相似的裁剪后的脸部进行推断。根据需要的推断时间，可以选择 OpenCV 或 CNN 人脸检测器。训练可以使用 Inception 或 ResNet。如果由于是视频而要求的推断时间少得多时，最好使用三个卷积，然后使用两个全连接层。请注意，年龄数据集通常存在巨大的类别不平衡，因此使用不同的度量指标（如精确率和召回率）会有所帮助。

2.6.4　微调服装模型

微调服装模型是一个不错的选择。具有多个分类属性的 softmax 层在这里会很有用。属性可以是模式、颜色等。

2.6.5　品牌安全

使用**支持向量机**（SVM）训练瓶颈层是一个很好的选择，因为图像在各个类中可能会有很大不同。这通常用于内容审核，以帮助避免显式的图像。你已经学会了如何解决图像分类中的新问题。

2.7　小结

本章介绍了用于训练分类任务的基本而有用的模型。我们学习了使用 Keras 和 TensorFlow API 的 MNIST 数据集的简单模型，了解了如何利用 TensorBoard 来观看训练过程，讨论了一些最新架构的具体应用，还介绍了几种提高准确率的方法，如数据增强、瓶颈层训练以及微调预训练的模型。此外还介绍了训练新模型的技巧和窍门。

下一章将介绍如何将深度学习模型可视化。我们会部署经过训练的模型进行推断，学习如何使用经过训练的层进行图像搜索，了解自编码器（autoencoder）的概念并将其用于特征的维数。

第 3 章

图像检索

深度学习也可以称为表示学习（representation learning），因为要在训练时学习模型中的特征或表示。在训练过程中隐藏层生成的**视觉特征**可用于计算距离度量指标。这些模型根据分类任务学习如何检测各层的边缘、模式等。

本章涉及以下主题：

- 如何从经过分类训练的模型中提取特征；
- 如何在生产系统中使用 TensorFlow Serving 进行更快的推断；
- 如何使用这些特征计算一个查询图像与一组目标之间的相似度；
- 使用分类模型进行排序；
- 如何提高检索系统的速度；
- 从整体上看系统的架构；
- 当目标图像过多时，使用自编码器学习一个紧凑描述子（descriptor）；
- 训练一个去噪自编码器。

3.1 理解视觉特征

深度学习模型经常因不可解释而受到批评。因为人们难以明晰深度学习模型的工作机理，基于神经网络的模型通常被认为是一个黑盒子。由于激活函数，深度学习模型各层对图像进行转换是非线性的，因此不易可视化。目前已经开发出来一些方法，通过将深度网络各层可视化来应对其不可解释性的批评。在本节中，我们将尝试将深层可视化，以理解模型的工作机理。

可以使用模型的激活和梯度实现可视化。通过以下技术激活可视化。

- **最近邻**：可以获取图像的层激活，并一起看到该激活最相近的图像。
- **降维**：可以通过主成分分析（principal component analysis，PCA）或 t-分布式随机邻域嵌入（t-distributed stochastic neighbor embedding，t-SNE）来降低激活的维度，以便在二维或三维中进行可视化。PCA 通过向最大方差方向投影值来减小维度。t-SNE 通过将最近点

映射到三个维度来减少维数。降维的具体使用和技术超出了本书的范围。为了解更多关于降维的信息，建议参考基本的机器学习资料。

维基百科是理解降维技术的一个好资料。

- **最大图块**：一个神经元被激活，具有最大激活值的相应图块被捕捉到。
- **遮挡**（occlusion）：图像在不同的位置被遮挡（阻挡），并且把激活值显示为热图（heat map），以理解图像的哪些部分是重要的。

在接下来的几节中，我们将看到如何实现这些特征的可视化。

3.1.1　深度学习模型的可视化激活

任何模型架构都可以通过任何层的滤波器进行可视化。使用该技术只能理解最初的一些层。使用最近邻方法对最后一层很有用。当用最近邻排列时，ImageNet 数据集如图 3-1 所示。



图　3-1

看着这幅图像，你可以看到相同的物体一起出现。有趣的是，像狗、猴子和猎豹这样的动物一起出现了，尽管它们没有在一个标签下训练。当对象相似时，图像的最近邻可视化是有用的，因此我们可以理解模型的预测。最后一层也可以通过降维技术（例如主成分分析和 t-SNE）进行可视化。我们将在下一节中看到使用降维的可视化实现。

3.1.2　嵌入可视化

使用 TensorBoard 可以在二维或三维中对嵌入层（最后一层的前一层）进行可视化。本节中的代码片段假定来自图像分类一章中训练后的卷积神经网络模型。首先，我们需要一个元数据文件，它是一个制表符分隔的文件。元数据文件的每一行都应该包含将要可视化的图像的标签。这里需要一个新变量来存储在会话创建和初始化之间定义的嵌入，如下所示：


```
no_embedding_data = 1000
embedding_variable = tf.Variable(tf.stack(
    mnist.test.images[:no_embedding_data], axis=0), trainable=False)
```

我们将采用 MNIST 测试数据并创建一个用于可视化的元数据文件，如下所示：

```
metadata_path = '/tmp/train/metadata.tsv'

with open(metadata_path, 'w') as metadata_file:
  for i in range(no_embedding_data):
  metadata_file.write('{}\n'.format(
  np.nonzero(mnist.test.labels[::1])[1:][0][i]))
```

通过设置上述代码中所示的参数，可以使嵌入变量不可训练。接下来，必须定义投影器配置：包括一个嵌入变量名称 `tensor_name`、元数据文件的路径和一个精灵图像（sprite image）。精灵图像是一个具有一些小图像的图像，用于表示用嵌入来可视化的标签。如下是定义嵌入的投影的代码：

```
from tensorflow.contrib.tensorboard.plugins import projector
projector_config = projector.ProjectorConfig()
embedding_projection = projector_config.embeddings.add()
embedding_projection.tensor_name = embedding_variable.name
embedding_projection.metadata_path = metadata_path
embedding_projection.sprite.image_path = os.path.join(work_dir +
'/mnist_10k_sprite.png')
embedding_projection.sprite.single_image_dim.extend([28, 28])
```

必须指定精灵图像维度，然后可以使用投影器通过汇总写入器和配置来可视化嵌入，如下所示：

```
projector.visualize_embeddings(train_summary_writer, projector_config)
tf.train.Saver().save(session, '/tmp/train/model.ckpt', global_step=1)
```

模型与会话一起保存。然后转到 TensorBoard 查看如图 3-2 所示的可视化。

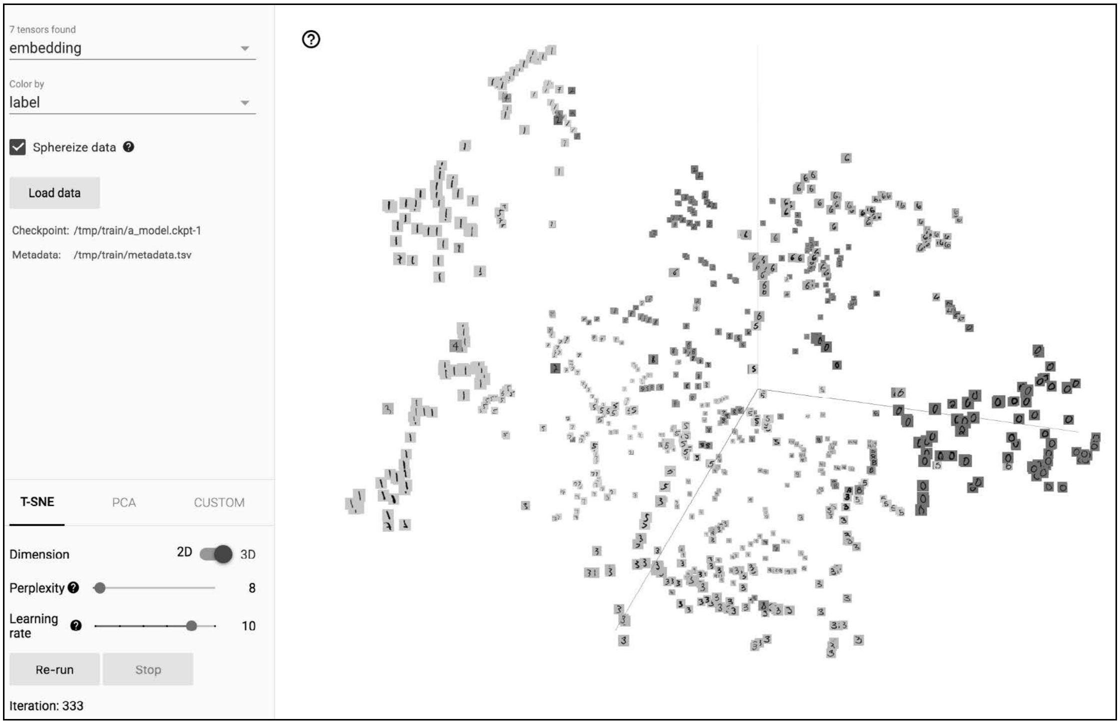


图　3-2

为获得与屏幕截图相似的可视化效果，需要用按钮选择 T-SNE 和颜色。你可以看到数字是如何一起出现的。这种可视化对于检查数据和经过训练的嵌入非常有用。这是 TensorBoard 的另一个强大功能。在下一节中，我们将实现可视化的引导式反向传播（guided backpropagation）。

引导式反向传播

直接可视化特征可能不会提供太多的信息。因此，我们使用反向传播的训练过程来激活滤波器，以实现更好的可视化。由于我们可以选择哪些神经元用于反向传播，因此称为引导式反向传播。在本节中，我们将实现引导式反向传播，将这些特征可视化。

我们将定义规模并加载 VGG 模型，如下所示：

```
image_width, image_height = 128, 128
vgg_model = tf.keras.applications.vgg16.VGG16(include_top=False)
```

这些层由一个字典组成，字典中层的名作为键，来自模型的层和权重作为键值以便于访问。现在我们将从第五个块 `block5_conv1` 开始得到第一个卷积层，用于计算可视化。输入和输出定义如下：

```
input_image = vgg_model.input
vgg_layer_dict = dict([(vgg_layer.name, vgg_layer) for vgg_layer in
vgg_model.layers[1:]])
vgg_layer_output = vgg_layer_dict['block5_conv1'].output
```

我们必须定义损失函数。损失函数将最大化特定层的激活值。因为这里试图最大化损失函数，所以这是一个梯度上升过程，而不是通常的梯度下降过程。对于梯度上升，平滑梯度很重要。所以我们通过归一化像素梯度来平滑这种情况下的梯度。这样的损失函数收敛得相当快。

应该对图像的输出进行归一化将其可视化回来，在优化过程中使用梯度上升来获得函数的最大值。我们可以通过定义评估器和梯度开始梯度上升优化。现在要定义损失函数并且计算梯度。迭代器在迭代过程中计算损失和梯度值，如下所示：


```
    filters = []
    for filter_idx in range(20):
    loss = tf.keras.backend.mean(vgg_layer_output[:, :, :, filter_idx])
    gradients = tf.keras.backend.gradients(loss, input_image)[0]
    gradient_mean_square =
tf.keras.backend.mean(tf.keras.backend.square(gradients))
    gradients /= (tf.keras.backend.sqrt(gradient_mean_square) + 1e-5)
    evaluator = tf.keras.backend.function([input_image], [loss, gradients])
```

输入是一个添加了一些噪声的随机灰度图像。生成随机图像并进行缩放，如下所示：

```
gradient_ascent_step = 1.
input_image_data = np.random.random((1, image_width, image_height, 3))
input_image_data = (input_image_data - 0.5) * 20 + 128
```

现在开始优化损失函数。对于某些滤波器，损失值可能为 0，这些应该忽略，如下所示：

```
for i in range(20):
    loss_value, gradient_values = evaluator([input_image_data])
    input_image_data += gradient_values * gradient_ascent_step
    # print('Loss :', loss_value)
    if loss_value <= 0.:
        break
```

在这个优化之后，通过均值减法和调整标准偏差完成归一化。然后，可以缩放滤波器并将其剪裁为梯度值，如下所示：

```
if loss_value > 0:
    filter = input_image_data[0]
    filter -= filter.mean()
    filter /= (filter.std() + 1e-5)
    filter *= 0.1
    filter += 0.5
    filter = np.clip(filter, 0, 1)
    filter *= 255
    filter = np.clip(filter, 0, 255).astype('uint8')
    filters.append((filter, loss_value))
```

这些滤波器是随机挑选出来的，并在这里可视化（图 3-3）。

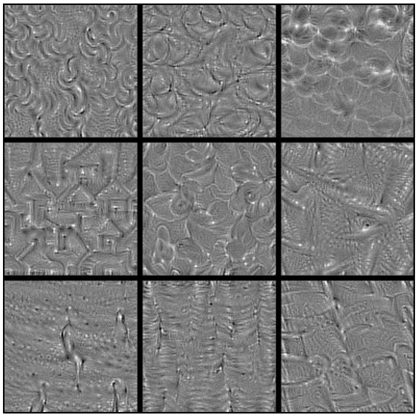
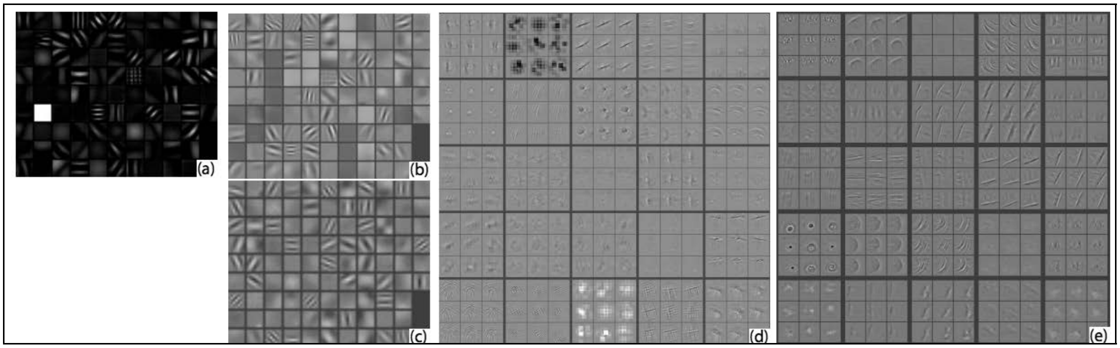


图　3-3

如图 3-3 所示的拼接图像并生成输出的代码与代码包一起提供。因为内容的感受野变得更大，所以可视化在后面的层变得复杂。一些滤波器只是旋转了，所以看起来相似。在这种情况下可以清楚地看到可视化的层次结构，如 Zeiler 等人所示。图 3-4 显示了不同层的直接可视化效果。



* 经 Zeiler 等人许可转载

图　3-4

前两层看起来像边缘和角落检测器。类似 Gabor 的滤波器只出现在第三层。Gabor 滤波器是线性的，传统上用于纹理分析。我们已经借助引导式反向传播并且直接看到了特征的可视化。接下来将介绍如何实现用于可视化的 DeepDream。

3.1.3　DeepDream

神经元激活可以在网络中的某一层放大，而不是合成图像。放大原始图像以查看特征效果的概念称为 DeepDream。创建 DeepDream 的步骤如下。

(1) 获取一幅图像，并从 CNN 中挑选一层。
(2) 在特定层上获取其激活。
(3) 修改梯度，使梯度和激活相等。
(4) 计算图像的梯度和反向传播。
(5) 图像必须进行抖动并使用正则归一化。
(6) 像素值应该裁剪。
(7) 为了达到分形的效果对图像进行多尺度处理。

首先导入相关的包：

```
import os
import numpy as np
import PIL.Image
import urllib.request
from tensorflow.python.platform import gfile
import zipfile
```

inception 模型是在 Imagenet 数据集和 Google 提供的模型文件上预训练的。我们可以下载该模型并将其用于该示例。下载并解压缩这个模型文件的 ZIP 存档到一个文件夹中，如下所示：

```
model_url =
'https://storage.googleapis.com/download.tensorflow.org/models/inception5h.zip'

file_name = model_url.split('/')[-1]

file_path = os.path.join(work_dir, file_name)

if not os.path.exists(file_path):
    file_path, _ = urllib.request.urlretrieve(model_url, file_path)
zip_handle = zipfile.ZipFile(file_path, 'r')
zip_handle.extractall(work_dir)
zip_handle.close()
```

这些命令应该在工作目录中创建三个新文件。该预训练的模型可以加载到会话中，如下所示：

```
graph = tf.Graph()
session = tf.InteractiveSession(graph=graph)
model_path = os.path.join(work_dir, 'tensorflow_inception_graph.pb')
with gfile.FastGFile(model_path, 'rb') as f:
    graph_defnition = tf.GraphDef()
    graph_defnition.ParseFromString(f.read())
```

一个会话从计算图的初始化开始。然后将已下载模型的计算图定义加载到内存中。作为预处理步骤，ImageNet 平均值必须从输入中减去。然后将预处理后的图像输入到计算图中：

```
input_placeholder = tf.placeholder(np.float32, name='input')
imagenet_mean_value = 117.0
preprocessed_input = tf.expand_dims(input_placeholder-imagenet_mean_value, 0)
tf.import_graph_def(graph_defnition, {'input': preprocessed_input})
```

现在会话和计算图已准备好进行推断。使用双线性插值将需要一个 resize_image 函数。可以添加一个 resize 函数方法，该方法使用 TensorFlow 会话调整图像的大小，如下所示：

```
def resize_image(image, size):
    resize_placeholder = tf.placeholder(tf.float32)
    resize_placeholder_expanded = tf.expand_dims(resize_placeholder, 0)
    resized_image = tf.image.resize_bilinear(resize_placeholder_expanded,
size)[0, :, :, :]
    return session.run(resized_image, feed_dict={resize_placeholder: image})
```

来自工作目录的图像可以加载到内存中并转换为浮点值，如下所示：

```
image_name = 'mountain.jpg'
image = PIL.Image.open(image_name)
image = np.float32(image)
```

加载的图像显示在这里作为参考（图 3-5）。



图　3-5

这里定义了倍频程（octave）数、大小和缩放比例：

```
no_octave = 4
scale = 1.4
window_size = 51
```

这些值适用于此处显示的示例。对于其他图像，需要根据其大小调整这些值。可以选择一个层用于“做梦”，该层的平均值将成为目标函数，如下所示：

```
score = tf.reduce_mean(objective_fn)
gradients = tf.gradients(score, input_placeholder)[0]
```

计算这些图像的梯度以进行优化。倍频程图像可以通过调整图像的大小来计算各种比例并找出差异，如下所示：

```
octave_images = []
for i in range(no_octave - 1):
    image_height_width = image.shape[:2]
    scaled_image = resize_image(image, np.int32(np.float32(image_height_width) /
scale))
    image_difference = image - resize_image(scaled_image, image_height_width)
    image = scaled_image
    octave_images.append(image_difference)
```


现在可以使用所有倍频程图像进行优化。窗口滑过图像，计算梯度激活以创建"梦境"，如下所示：

```
for octave_idx in range(no_octave):
    if octave_idx > 0:
        image_difference = octave_images[-octave_idx]
        image = resize_image(image, image_difference.shape[:2]) +
image_difference

    for i in range(10):
        image_heigth, image_width = image.shape[:2]
        sx, sy = np.random.randint(window_size, size=2)
        shifted_image = np.roll(np.roll(image, sx, 1), sy, 0)
        gradient_values = np.zeros_like(image)

        for y in range(0, max(image_heigth - window_size // 2, window_size),
window_size):
            for x in range(0, max(image_width - window_size // 2, window_size),
window_size):
                sub = shifted_image[y:y + window_size, x:x + window_size]
                gradient_windows = session.run(gradients, {input_placeholder: sub})
                gradient_values[y:y + window_size, x:x + window_size]= gradient_windows

        gradient_windows = np.roll(np.roll(gradient_values, -sx, 1), -sy, 0)
        image += gradient_windows * (1.5 / (np.abs(gradient_windows).mean() + 1e-7))
```

现在，创建 DeepDream 的优化已完成，并可通过剪切值保存，如下所示：

```
image /= 255.0
image = np.uint8(np.clip(image, 0, 1) * 255)
PIL.Image.fromarray(image).save('dream_' + image_name, 'jpeg')
```

在本节中，我们看到了创建 DeepDream 的过程。结果如图 3-6 所示。



图　3-6

正如所见，到处是激活的犬蛞蝓（dog slu）。我们可以尝试其他层并查看结果。这些结果可以用于艺术目的。同样，可以激活其他层以产生不同的伪像。下一节将介绍一些可以欺骗深度学习模型的对抗样本。

3.1.4　对抗样本

图像分类算法已经在几个数据集上达到了人类级别的准确率。但他们很容易被对抗样本欺骗。对抗样本是合成的图像，它通过欺骗一个模型来产生需要的结果。取任意图像并选择一个不正确的随机的目标类，如图 3-7 所示。这幅图像可以用噪声修改，直到网络被欺骗。图 3-7 显示了对该模型进行对抗攻击的例子。



* 经 Goodfellow 等人许可转载

图　3-7

在该图中，左侧显示的图像具有特定标签的 58%置信度。左侧的图像与中间显示的噪声结合在一起形成右侧的图像。对于人来说，有噪声的图像看起来是一样的。但带有噪声的图像可以用 97%的置信度以不同的标签进行预测。尽管图像具有非常不同的对象，但要将高置信度分配给一个特定的示例。这是深度学习模型的一个问题，因此，你应该明白这适用于何处。

- ❑ 对抗样本甚至可以在没有访问模型的情况下生成。你可以训练自己的模型，产生一个对抗样本，并且仍然可以欺骗另外一个不同的模型。

- 虽然在实际中很少发生，但当有人试图欺骗系统来发送垃圾邮件或导致崩溃时，它就成了一个真正的问题。
- 所有的机器学习模型都容易出现这个问题，而不仅仅是深度学习模型。

考虑到对抗样本，你应该了解在安全关键系统上部署深度学习模型的后果。下一节将介绍如何利用 TensorFlow Serving 获得更快的推断。

3.2 模型推断

任何新数据都可以传递给模型以获得结果。从图像中获取分类结果或特征的过程称为**推断**。训练和推断通常发生在不同的计算机上，并在不同的时间执行。我们将学习存储一个模型、执行推断并使用 TensorFlow Serving 作为具有良好延迟和吞吐量的服务器。

3.2.1 导出模型

训练后的模型必须导出并保存。权重、偏差和图都要存储下来以供推断。我们将训练一个 MNIST 模型并存储它。使用以下代码，从定义所需的常量开始：

```
work_dir = '/tmp'
model_version = 9
training_iteration = 1000
input_size = 784
no_classes = 10
batch_size = 100
total_batches = 200
```

`model_version` 可以是一个整数，用于指定我们要为服务导出哪个模型。feature config 存储为一个带有占位符名称及其相应数据类型的字典。预测的类和标签都被映射好。类别占位符可以与 API 一起使用。

```
tf_example = tf.parse_example(tf.placeholder(tf.string, name='tf_example'),
                              {'x': tf.FixedLenFeature(shape=[784],
dtype=tf.float32), })
x_input = tf.identity(tf_example['x'], name='x')
```

使用下面的代码，可以用权重、偏差、logits 和优化器来定义一个简单的分类器：

```
y_input = tf.placeholder(tf.float32, shape=[None, no_classes])
weights = tf.Variable(tf.random_normal([input_size, no_classes]))
bias = tf.Variable(tf.random_normal([no_classes]))
logits = tf.matmul(x_input, weights) + bias
softmax_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=y_input,
logits=logits)
loss_operation = tf.reduce_mean(softmax_cross_entropy)
optimiser = tf.train.GradientDescentOptimizer(0.5).minimize(loss_operation)
```

如下所示训练这个模型：

```
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
for batch_no in range(total_batches):
    mnist_batch = mnist.train.next_batch(batch_size)
    _, loss_value = session.run([optimiser, loss_operation], feed_dict={
        x_input: mnist_batch[0],
        y_input: mnist_batch[1]
    })
    print(loss_value)
```

定义预测签名并导出该模型。将该模型保存到持久性存储中，以便可以在稍后的时间点进行推断。这通过反序列化导出数据并将其存储为不同系统可以理解的格式。具有不同变量和占位符的多个图可用于导出。它还支持 `signature_defs` 和 assets。`signature_defs` 具有指定的输入和输出，因为输入和输出将从外部客户端访问。assets 是用于推断的非图的组件，例如词汇表等。

分类签名使用对 TensorFlow 的分类 API 的访问。一个输入是强制性的，并且有两个可选输出（预测类别和预测概率），其中至少一个是强制性的。预测签名为输入和输出的数量提供了灵活性。可以定义多个输出并从客户端明确查询。`signature_def` 显示在这里：

```
signature_def = (
      tf.saved_model.signature_def_utils.build_signature_def(
          inputs={'x': tf.saved_model.utils.build_tensor_info(x_input)},
          outputs={'y': tf.saved_model.utils.build_tensor_info(y_input)},
          method_name="tensorflow/serving/predict"))
```

最后，将 metagraph 和变量添加到具有预测签名的构建器中：

```
model_path = os.path.join(work_dir, str(model_version))
saved_model_builder = tf.saved_model.builder.SavedModelBuilder(model_path)
saved_model_builder.add_meta_graph_and_variables(
      session, [tf.saved_model.tag_constants.SERVING],
      signature_def_map={
          'prediction': signature_def
      },
      legacy_init_op=tf.group(tf.tables_initializer(), name='legacy_init_op'))
saved_model_builder.save()
```

构建器已保存并准备好被服务器使用。所示的示例适用于任何模型，可用于导出。在下一节中，我们将提供并查询导出的模型。

3.2.2 提供训练好的模型

上一节导出的模型可以通过 TensorFlow Serving 提供，使用以下命令：

```
tensorflow_model_server --port=9000 --model_name=mnist --
model_base_path=/tmp/mnist_model/
```

model_base_path 指向导出模型的目录。服务器现在可以通过客户端进行测试。请注意，这不是一个 HTTP 服务器。因此，需要用此处所示的客户端，而不是 HTTP 客户端。导入所需的库：

```
from grpc.beta import implementations
import numpy
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from tensorflow_serving.apis import predict_pb2
from tensorflow_serving.apis import prediction_service_pb2
```

为并发、测试次数和工作目录添加常量的声明。定义一个类用于计算返回的结果。定义一个**远程过程调用**（remote procedure call，RPC）的回调，其中使用一个计数器计算预测值，如下所示：


```
concurrency = 1
num_tests = 100
host = ''
port = 8000
work_dir = '/tmp'


def _create_rpc_callback():
  def _callback(result):
      response = numpy.array(
        result.result().outputs['y'].float_val)
      prediction = numpy.argmax(response)
      print(prediction)
  return _callback
```

根据你的要求修改主机和端口。callback 方法定义了响应从服务器返回时所需的步骤。在这个例子中，计算了概率的最大值。通过调用服务器运行推断：

```
test_data_set = mnist.test
test_image = mnist.test.images[0]

predict_request = predict_pb2.PredictRequest()
predict_request.model_spec.name = 'mnist'
predict_request.model_spec.signature_name = 'prediction'

predict_channel = implementations.insecure_channel(host, int(port))
predict_stub =
prediction_service_pb2.beta_create_PredictionService_stub(predict_channel)

predict_request.inputs['x'].CopyFrom(
    tf.contrib.util.make_tensor_proto(test_image, shape=[1, test_image.size]))
result = predict_stub.Predict.future(predict_request, 3.0)
result.add_done_callback(
    _create_rpc_callback())
```

重复调用推断来评估准确率、延迟和吞吐量。推断的错误率应该在 90%左右，并发性应该很高。为从模型中获取结果和特征，导出和客户端方法可以一起使用。在下一节中，我们将构建检索流水线。

3.3　基于内容的图像检索

基于内容的图像检索（content-based image retrieval，CBIR）技术将查询图像作为输入，并对来自目标图像数据库的图像进行排序来产生输出。CBIR 是一个具有特定目标的图像搜索引擎。为了检索任务，一个目标图像的数据库是必需的。返回距查询图像的距离最小的目标图像。我们可以直接使用图像进行相似性处理，但问题如下：

- 图像的尺寸很大；
- 像素有很多冗余；
- 像素不包含语义信息。

因此，我们训练一个目标分类模型，并使用模型中的特征进行检索。然后通过相同的模型传递查询图像和目标数据库来获取特征。这些模型也可以称为**编码器**，因为它们为特定任务进行图像信息编码。编码器应该能够捕捉全局和局部特征。我们可以使用第 2 章中研究的模型进行分类任务的训练。由于蛮力搜索或线性扫描速度较慢，搜索图像可能需要很长时间。因此，需要一些快速检索的方法。以下是一些快速匹配的方法。

- **局部敏感散列**（locality sensitive hashing，LSH）：LSH 将特征投影到它们的子空间，并且可以为候选者提供一个列表，并在稍后进行精细特征排序。和本章前面介绍的 PCA 和 t-SNE 相似，这也是一种降维技术。它在较低维度具有特征容器。
- **多索引散列**（multi-index hashing）：这种方法对特征进行散列，就像鸽洞匹配一样，它使用汉明距离来加快计算速度。汉明距离是用二进制表示的数的位置差异数。

这些方法速度更快，只需要较少的内存，并具有准确性的权衡。这些方法也不能捕获语义差异。为了根据查询获得更好的结果，可以对匹配结果进行重排序。重排序可以通过将返回的目标图像重新排序来改善结果。重排序可能使用的技术如下。

- **几何验证**（geometric verification）：该方法仅依据返回的相似几何部分将几何图形和目标图像匹配。
- **查询扩展**（query expansion）：该方法扩大了目标图像的列表并且进行了彻底搜索。
- **相关反馈**（relevance feedback）：该方法从使用中获得反馈并返回结果。基于用户输入，将完成重排序。

这些技术已经很好地用于文本，也可以用于图像。在本章中，我们将重点介绍提取特征并将其用于 CBIR。

3.3.1　构建检索流水线

从查询图像的目标图像中获得最佳匹配的步骤序列称为**检索流水线**（retrieval pipeline）。检

索流水线有多个步骤或组件。图像数据库的特征必须离线提取并存储在一个数据库中。每个查询图像都必须提取特征，并且在所有目标图像上计算相似度。然后可以对图像进行排序以获得最终输出。检索流水线如图 3-8 所示。

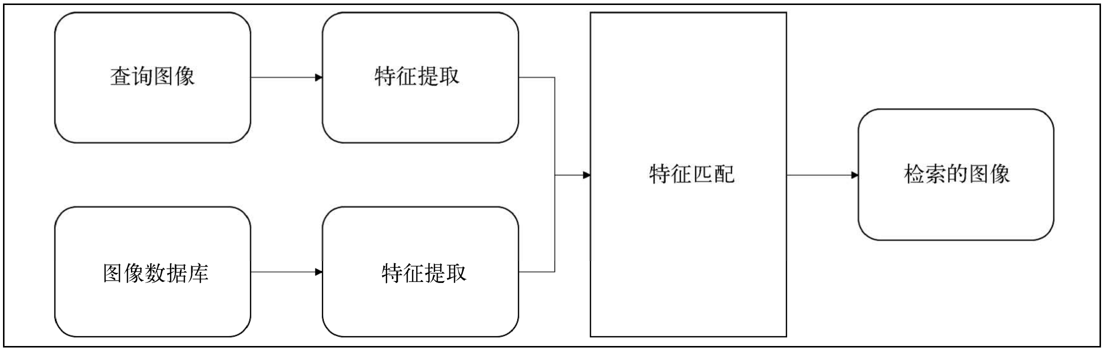


图　3-8

特征提取步骤必须快速，可以使用 TensorFlow Serving 做到这一点。我们可以根据应用选择使用哪些特征。例如，当需要基于纹理的匹配时，可以使用初始层；当需要在物体级别匹配时，可以使用后面的层。下一节将介绍如何从预训练的 inception 模型中提取特征。

1. 提取图像的瓶颈特征

瓶颈特征是在预分类层中计算的值。在本节中，我们将介绍如何使用 TensorFlow 从预训练的模型中提取瓶颈特征。使用以下代码，从导入所需的库开始：

```
import os
import urllib.request
from tensorflow.python.platform import gfile
import tarfile
```

然后，我们需要下载带有图定义及其权重的预训练的模型。TensorFlow 已经使用 inception 架构在 ImageNet 数据集上训练了一个模型并提供了该模型。我们将下载此模型并使用以下代码将其解压缩到本地文件夹中：

```
model_url =
'http://download.tensorflow.org/models/image/imagenet/inception-2015-12-05.tgz'
file_name = model_url.split('/')[-1]
file_path = os.path.join(work_dir, file_name)

if not os.path.exists(file_path):
    file_path, _ = urllib.request.urlretrieve(model_url, file_path)
tarfile.open(file_path, 'r:gz').extractall(work_dir)
```

这创建了一个文件夹，只有当它不存在时才下载模型。如果代码重复执行，则模型不会每次都下载。该图以**协议缓冲区**（Protocol Buffer，protobuf）格式存储在文件中。它必须作为字符串

读取并传递给 tf.GraphDef()对象，以将它引入内存中：

```
model_path = os.path.join(work_dir, 'classify_image_graph_def.pb')
with gfile.FastGFile(model_path, 'rb') as f:
    graph_defnition = tf.GraphDef()
    graph_defnition.ParseFromString(f.read())
```

在 inception 模型中，瓶颈层名为 pool_3/_reshape：0，层为 2048 个维度。输入占位符名为 DecodeJpeg/contents：0，调整大小张量名为 ResizeBilinear：0。我们可以使用 tf.import_graph_def 来导入图的定义和进一步操作所需的返回张量：

```
bottleneck, image, resized_input = (
    tf.import_graph_def(
        graph_defnition,
        name='',
        return_elements=['pool_3/_reshape:0',
                         'DecodeJpeg/contents:0',
                         'ResizeBilinear:0'])
)
```

获得一个查询和目标图像并将它加载到内存中。gfile 函数提供了一种能够更快地将图像加载到内存中的方法。

```
query_image_path = os.path.join(work_dir, 'cat.1000.jpg')
query_image = gfile.FastGFile(query_image_path, 'rb').read()
target_image_path = os.path.join(work_dir, 'cat.1001.jpg')
target_image = gfile.FastGFile(target_image_path, 'rb').read()
```

让我们定义一个函数，它使用 session 和 image 从图像中提取瓶颈特征：

```
def get_bottleneck_data(session, image_data):
    bottleneck_data = session.run(bottleneck, {image: image_data})
    bottleneck_data = np.squeeze(bottleneck_data)
    return bottleneck_data
```

为从预训练的模型中获取瓶颈值，启动会话并传递图像以运行前向推断：

```
query_feature = get_bottleneck_data(session, query_image)
print(query_feature)
target_feature = get_bottleneck_data(session, target_image)
print(target_feature)
```

运行以上代码应该打印如下内容：

```
[ 0.55705792 0.36785451 1.06618118 ..., 0.6011821 0.36407694
  0.0996572 ]
[ 0.30421323 0.0926369 0.26213276 ..., 0.72273785 0.30847171
  0.08719242]
```

为获得更多目标图像，这个计算特征的过程可以按比例缩放。使用这些值，可以计算查询图像和目标数据库之间的相似度，如下节所述。

2. 计算查询图像和目标数据库的相似度

NumPy 的 `linalg.norm` 对计算**欧几里得距离**很有用。通过计算图像之间的欧几里得距离，可以得出查询图像和目标数据库之间的相似度，如下所示：

```
dist = np.linalg.norm(np.asarray(query_feature) - np.asarray(target_feature))
print(dist)
```

运行此命令应该打印以下内容：

```
16.9965
```

这是可用于相似度计算的指标。查询图像与目标图像之间的欧几里得距离越小，两幅图像越相似。因此，欧几里得距离是相似度的度量。使用这些特征来计算欧几里得距离基于这样的假设：特征是在训练模型期间学习到的。数百万幅图像这样大规模的计算效率不高。生产系统预期以毫秒为单位返回结果。下一节将探讨如何提高检索效率。

3.3.2 有效的检索

检索可能会很慢，因为它是一种蛮力方法。使用近似最近邻方法可以使匹配更快。维度灾难也会有影响，如图 3-9 所示。

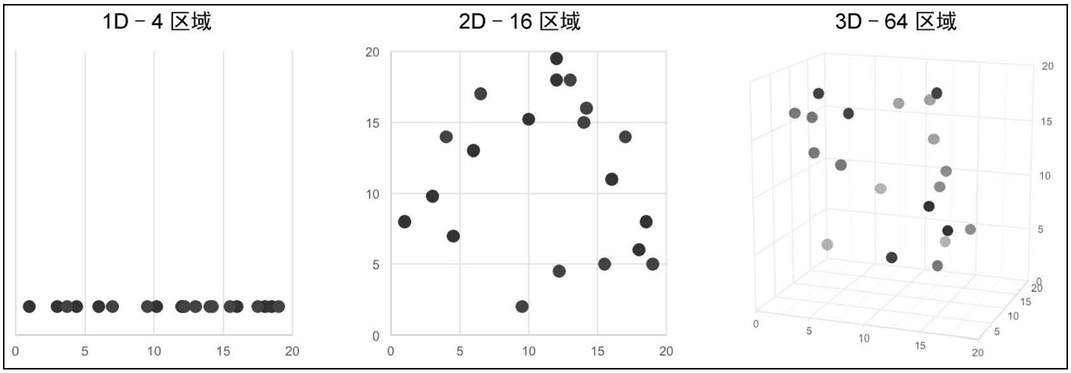


图 3-9

随着维度不断增加，复杂度从二维增加到三维，距离的计算也变得更慢。为了加快距离搜索，将在下一节介绍一个近似方法。

1. 使用近似最近邻进行更快匹配

ANNOY（approximate nearest neighbour oh yeah）是一个能够快速进行最近邻搜索的方法。ANNOY 通过随机投影来构建树。树结构可以让最接近的匹配项更容易找到。你可以创建一个 `ANNOYIndex` 来更快地检索，如下所示：

```
def create_annoy(target_features):
    t = AnnoyIndex(layer_dimension)
    for idx, target_feature in enumerate(target_features):
        t.add_item(idx, target_feature)
    t.build(10)
    t.save(os.path.join(work_dir, 'annoy.ann'))

create_annoy(target_features)
```

特征的维度是创建索引所必需的。然后将这些项添加到索引中，并构建树。通过权衡时间和空间的复杂性，树的数量越大，结果越精确。可以创建索引并加载到内存中。ANNOY 可以被查询，如下所示：

```
annoy_index = AnnoyIndex(10)
annoy_index.load(os.path.join(work_dir, 'annoy.ann'))
matches = annoy_index.get_nns_by_vector(query_feature, 20)
```

匹配列表可用于检索图像细节。这些项目的索引将被返回。

> 请访问 https://github.com/spotify/annoy，了解 ANNOY 的完整实现方案及与其他近似最近邻算法在准确率和速度方面的基准比较。

- **ANNOY 的优势**

使用 ANNOY 有很多原因。其主要优点如下。

- 有一个内存映射的数据结构，因此 RAM 占用较低。由于这一点，可以在多个进程之间共享同一个文件。
- 可以使用曼哈顿距离、余弦距离或欧几里得距离等多种距离来计算查询图像与目标数据库之间的相似度。

2. 原始图像的自编码器

自编码器是用于生成高效编码的无监督算法。输入层和目标输出通常是相同的。在减少和增加之间的层如图 3-10 所示。

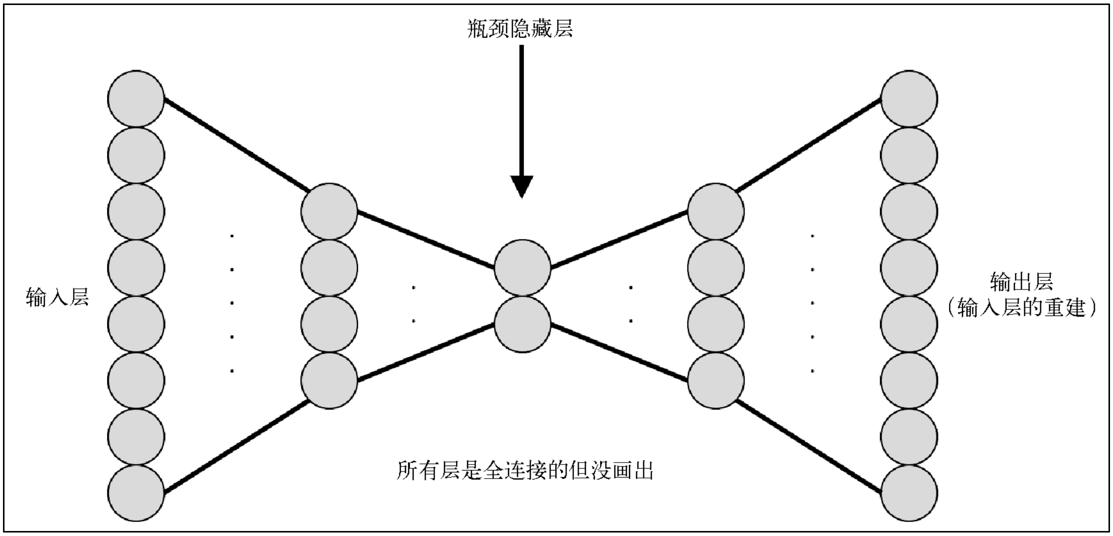


图　3-10

瓶颈层是一个减小了维度的中间层。瓶颈层的左侧称为**编码器**，右侧称为**解码器**。编码器通常会减少数据的维度，而解码器会增加维度。编码器和解码器的组合称为自编码器。整个网络用重建错误来训练。理论上，可以存储瓶颈层，并且解码器网络可以重建原始数据。这样减少了维度，并且可以很容易地进行编程。使用如下代码定义卷积、反卷积和全连接层：

```
def fully_connected_layer(input_layer, units):
    return tf.layers.dense(
        input_layer,
        units=units,
        activation=tf.nn.relu
    )

def convolution_layer(input_layer, filter_size):
    return tf.layers.conv2d(
        input_layer,
        filters=filter_size,
        kernel_initializer=tf.contrib.layers.xavier_initializer_conv2d(),
        kernel_size=3,
        strides=2
    )

def deconvolution_layer(input_layer, filter_size, activation=tf.nn.relu):
    return tf.layers.conv2d_transpose(
        input_layer,
        filters=filter_size,
        kernel_initializer=tf.contrib.layers.xavier_initializer_conv2d(),
        kernel_size=3,
        activation=activation,
        strides=2
    )
```

用五层卷积定义收敛的编码器，如下所示：

```
input_layer = tf.placeholder(tf.float32, [None, 128, 128, 3])
convolution_layer_1 = convolution_layer(input_layer, 1024)
convolution_layer_2 = convolution_layer(convolution_layer_1, 512)
convolution_layer_3 = convolution_layer(convolution_layer_2, 256)
convolution_layer_4 = convolution_layer(convolution_layer_3, 128)
convolution_layer_5 = convolution_layer(convolution_layer_4, 32)
```

通过展平第五卷积层来计算瓶颈层。为适合卷积层，瓶颈层再次被重建回来，如下所示：

```
convolution_layer_5_flattened = tf.layers.flatten(convolution_layer_5)
bottleneck_layer = fully_connected_layer(convolution_layer_5_flattened, 16)
c5_shape = convolution_layer_5.get_shape().as_list()
c5f_flat_shape = convolution_layer_5_flattened.get_shape().as_list()[1]
fully_connected = fully_connected_layer(bottleneck_layer, c5f_flat_shape)
fully_connected = tf.reshape(fully_connected,
                             [-1, c5_shape[1], c5_shape[2], c5_shape[3]])
```

计算可重建图像的发散或解码器部分，如下所示：

```
deconvolution_layer_1 = deconvolution_layer(fully_connected, 128)
deconvolution_layer_2 = deconvolution_layer(deconvolution_layer_1, 256)
deconvolution_layer_3 = deconvolution_layer(deconvolution_layer_2, 512)
deconvolution_layer_4 = deconvolution_layer(deconvolution_layer_3, 1024)
deconvolution_layer_5 = deconvolution_layer(deconvolution_layer_4, 3,
                                            activation=tf.nn.tanh)
```

这个网络经过训练并迅速收敛。当图像特征传递过来时，瓶颈层可以被存储。这有助于减小数据库的大小，然后可用于检索。我们只需要编码器部分来索引这些特征。自编码器是一种有损压缩算法。它与其他压缩算法不同，因为它从数据中学习压缩模式。因此，一个自编码器模型是特定于数据的。为获得更好的可视化，自编码器可以与 t-SNE 相结合。自编码器学到的瓶颈层可能对其他任务没有用处。瓶颈层的大小可能比以前的层更大。在这种情况下，发散和收敛的连接是稀疏的自编码器。在下一节中，我们将学习自编码器的另一个应用。

3.3.3 使用自编码器去噪

自编码器也可用于图像去噪。去噪是从图像中去除噪声的过程。去噪编码器可以用无监督的方式进行训练。在正常图像中引入噪声，并且用自编码器针对原始图像进行训练。之后，完整的自编码器可用于产生无噪声图像。本节将提供 MNIST 图像去噪的逐步指导。导入所需的库并定义占位符，如下所示：

```
x_input = tf.placeholder(tf.float32, shape=[None, input_size])
y_input = tf.placeholder(tf.float32, shape=[None, input_size])
```

因为在一个自编码器中，`x_input` 和 `y_input` 的形状相同。然后，定义一个密集层，默认激活为 `tanh` 激活函数。`add_variable_summary` 方法是从第 2 章示例中导入的。下面显示了密集层的定义：

```
def dense_layer(input_layer, units, activation=tf.nn.tanh):
    layer = tf.layers.dense(
        inputs=input_layer,
        units=units,
        activation=activation
    )
    add_variable_summary(layer, 'dense')
    return layer
```

接下来，可以定义自编码器层。这个自编码器只有全连接层。编码器部分有减少维度的三个层。解码器部分有增加维度的三个层。编码器和解码器都是对称的，如下所示：

```
layer_1 = dense_layer(x_input, 500)
layer_2 = dense_layer(layer_1, 250)
layer_3 = dense_layer(layer_2, 50)
layer_4 = dense_layer(layer_3, 250)
layer_5 = dense_layer(layer_4, 500)
layer_6 = dense_layer(layer_5, 784)
```

隐藏层的维度是任意选择的。接下来，定义 `loss` 和 `optimizer`。这里使用 `sigmoid` 而不是 `softmax` 作为分类，如下所示：

```
with tf.name_scope('loss'):
    softmax_cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
        labels=y_input, logits=layer_6)
    loss_operation = tf.reduce_mean(softmax_cross_entropy,name='loss')
    tf.summary.scalar('loss', loss_operation)

with tf.name_scope('optimiser'):
    optimiser = tf.train.AdamOptimizer().minimize(loss_operation)
```

TensorBoard 提供了另一种名为 `image` 的汇总，它对于图像的可视化非常有用。我们将接受输入 `layer_6` 并重新整形它，以将其添加到汇总中，如下所示：

```
x_input_reshaped = tf.reshape(x_input, [-1, 28, 28, 1])
tf.summary.image("noisy_images", x_input_reshaped)

y_input_reshaped = tf.reshape(y_input, [-1, 28, 28, 1])
tf.summary.image("original_images", y_input_reshaped)

layer_6_reshaped = tf.reshape(layer_6, [-1, 28, 28, 1])
tf.summary.image("reconstructed_images", layer_6_reshaped)
```

图像的数量默认限制为三个，这可以更改。这是为了限制它将所有图像写入汇总文件夹。接下来，将所有汇总合并，并将该图添加到汇总写入器，如下所示：

```
merged_summary_operation = tf.summary.merge_all()
train_summary_writer = tf.summary.FileWriter('/tmp/train', session.graph)
```

正常的随机噪声可以添加到图像并作为输入张量传入。噪声添加后，额外的值被剪切。目标是自己的原始图像。添加噪声和训练过程如下所示：

```
    for batch_no in range(total_batches):
        mnist_batch = mnist_data.train.next_batch(batch_size)
        train_images, _ = mnist_batch[0], mnist_batch[1]
        train_images_noise = train_images + 0.2 *
    np.random.normal(size=train_images.shape)
        train_images_noise = np.clip(train_images_noise, 0., 1.)
        _, merged_summary = session.run([optimiser, merged_summary_operation],
                                       feed_dict={
            x_input: train_images_noise,
            y_input: train_images,
        })
        train_summary_writer.add_summary(merged_summary, batch_no)
```

当这个训练开始时，结果可以在 TensorBoard 中看到。损失如图 3-11 所示。

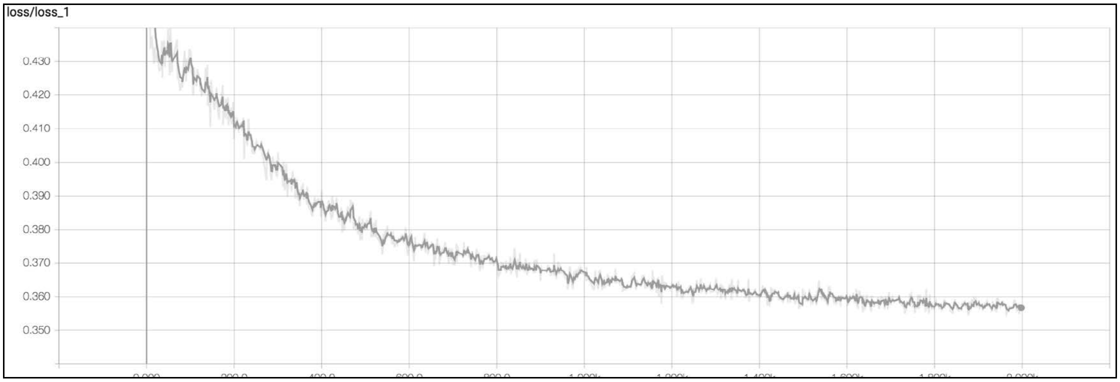


图　3-11

损失稳步下降，在迭代过程中将继续缓慢下降。这显示了自编码器是如何快速收敛的。接下来，从原始图像显示三位数字，如图 3-12 所示。



图　3-12

图 3-13 是添加了噪声的相同图像。



图　3-13

你会注意到有很大的噪声，而且它作为了一个输入。接下来，使用去噪自编码器重建相同数字的图像，如图 3-14 所示。

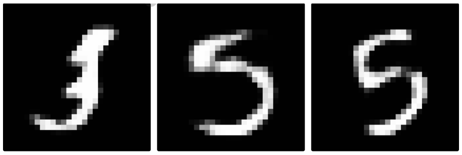


图 3-14

你会注意到去噪自编码器在消除噪声方面做得非常出色。可以在测试图像上运行此功能，并且可以维持质量。更复杂的数据集可以使用卷积神经网络获得更好的结果。因为是以无监督方式进行训练的，这个例子展示了计算机视觉深度学习的能力。

3.4 小结

在本章中，我们学习了如何从图像中提取特征并将其用于 CBIR，如何使用 TensorFlow Serving 来获取图像特征的推断，如何利用近似最近邻或快速匹配方法而不是线性扫描方法来检索，以及如何使用散列改善结果。本章引入了自编码器的思路，并且介绍了如何训练较小的特征向量以进行搜索，还给出了使用自编码器进行图像去噪的示例。我们看到了使用基于位的比较的可能性，这种比较方法可以将其扩展至数十亿的图像。

下一章将介绍如何训练目标检测问题的模型。我们将利用开源模型来获得良好的准确性并理解背后的所有算法，并使用所有的思路来训练行人检测模型。

第4章 目标检测

目标检测是在一幅图像中查找目标位置的行为。本章将介绍目标检测技术并实现行人检测。

本章涉及以下主题：

- 基础知识以及定位与检测的区别；
- 各种数据集及其描述；
- 用于目标定位和检测的算法；
- 用于目标检测的 TensorFlow API；
- 训练新的目标检测模型；
- 用 YOLO 算法在移动车辆上进行行人检测。

4.1 检测图像中的目标

近年来，目标检测在应用和研究方面都出现了爆炸性增长。目标检测是计算机视觉中的重要问题。与图像分类任务类似，深度网络在检测中表现出了更好的性能。目前，这些技术的准确性非常好，故在许多应用中都有使用。

图像分类将图像作为整体打标签。除了标记目标之外，还需要查找目标的位置，这称为**目标定位**（object localization）。通常，目标的位置由直角坐标值定义。用直角坐标值在图像中找到多个目标称为检测。图 4-1 是一个目标检测的例子。



图 4-1

该图像显示了四个带边界框的目标。我们将学习可以执行查找边界框任务的算法。在机器人视觉方面，它的应用非常多，例如自动驾驶汽车和工业目标。定位和检测任务可以总结为以下两点：

- 定位是检测图像中带有一个给定标签的单个目标；
- 检测是找到图像中带有给定标签的所有目标。

两者的不同之处是目标的数量。在检测中，目标的数量是可变的。在设计用于定位或检测的深度学习模型的架构时，这种小区别会导致很大的差异。接下来，我们将看到可用于这些任务的各种数据集。

4.2 探索数据集

可用于目标定位和检测的数据集有很多。在本节中，我们将探索研究团体用来评估算法的数据集。这些数据集中包含数量不等的目标，标定的数量范围从 20 到 200 个不等。这使目标检测变得困难。某些数据集在一幅图像中的目标太多，而某些数据集中每幅图像却只有一个目标。接下来，我们将详细查看一些数据集。

4.2.1 ImageNet 数据集

ImageNet 拥有评估用于分类、定位和检测任务的数据。第 2 章详细讨论了分类数据集。与分类数据类似，定位任务有 1000 个分类。准确率是根据最高五项检测结果计算出来的。所有图像中至少有一个边界框。对 200 个目标的检测问题有 470 000 幅图像，平均每幅图像有 1.1 个对象。

4.2.2 PASCAL VOC 挑战

PASCAL VOC 挑战在 2005 年至 2012 年间展开。这一挑战被认为是目标检测技术的基准。该数据集中有 20 个分类，包含 11 530 幅用于训练和验证的图像，其中感兴趣的区域有 27 450 个标定。以下是数据集中的 20 个分类。

- 人：人。
- 动物：鸟、猫、牛、狗、马、羊。
- 车辆：飞机、自行车、船、巴士、汽车、摩托车、火车。
- 室内物品：瓶子、椅子、餐桌、盆栽植物、沙发、电视/监视器。

该数据集平均每幅图像有 2.4 个目标。

4.2.3 COCO 目标检测挑战

COCO（common objects in context）数据集包含 20 万幅图像。80 个类别中有超过 50 万个目标标定。它是最广泛公开的目标检测数据库。图 4-2 展示了数据集中存在的目标列表。

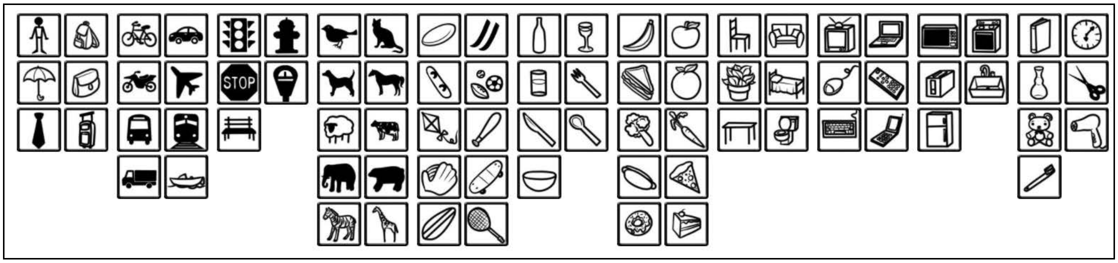


图　4-2

平均每幅图像的目标数为 7.2。这些是目标检测挑战的著名数据集。接下来，我们将学习如何评估针对这些数据集的算法。

4.2.4　使用指标评估数据集

指标对于理解深度学习任务至关重要。由于人的标定，目标检测和定位的度量指标是独特的。人可能标定了一个被称为**真实值**（ground-truth）的框。真实值不一定是绝对的事实。而且，不同人标定的框可能相差几个像素。因此，算法难以检测人绘制的确切边界框。**交并比**（intersection over union，IoU）用于评估定位任务。**平均精度**（mean average precision，mAP）用于评估检测任务。我们将在以下几节中看到这些指标的描述。

1. 交并比

交并比是真实值和预测面积重叠部分的面积与总面积的比值。图 4-3 是该指标的视觉解释。

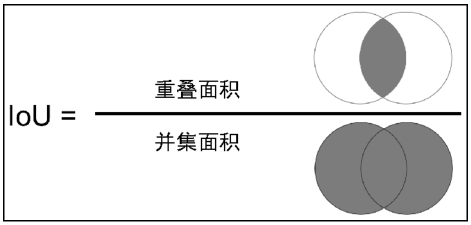


图　4-3

实际计算时，用两个方块代表真实值和预测值的边界框。交并比通过重叠面积与并集面积的比来计算。下面是计算真实值和预测值边界框的交并比的脚本：

```
def calculate_iou(gt_bb, pred_bb):
    '''
    :param gt_bb: ground truth bounding box
    :param pred_bb: predicted bounding box
    '''
    gt_bb = tf.stack([
        gt_bb[:, :, :, :, 0] - gt_bb[:, :, :, :, 2] / 2.0,
```

```
        gt_bb[:, :, :, :, 1] - gt_bb[:, :, :, :, 3] / 2.0,
        gt_bb[:, :, :, :, 0] + gt_bb[:, :, :, :, 2] / 2.0,
        gt_bb[:, :, :, :, 1] + gt_bb[:, :, :, :, 3] / 2.0])
    gt_bb = tf.transpose(gt_bb, [1, 2, 3, 4, 0])
    pred_bb = tf.stack([
        pred_bb[:, :, :, :, 0] - pred_bb[:, :, :, :, 2] / 2.0,
        pred_bb[:, :, :, :, 1] - pred_bb[:, :, :, :, 3] / 2.0,
        pred_bb[:, :, :, :, 0] + pred_bb[:, :, :, :, 2] / 2.0,
        pred_bb[:, :, :, :, 1] + pred_bb[:, :, :, :, 3] / 2.0])
    pred_bb = tf.transpose(pred_bb, [1, 2, 3, 4, 0])
    area = tf.maximum(
        0.0,
        tf.minimum(gt_bb[:, :, :, :, 2:], pred_bb[:, :, :, :, 2:]) -
        tf.maximum(gt_bb[:, :, :, :, :2], pred_bb[:, :, :, :, :2]))
    intersection_area = area[:, :, :, :, 0] * area[:, :, :, :, 1]
    gt_bb_area = (gt_bb[:, :, :, :, 2] - gt_bb[:,  :, :, :, 0]) * \
                 (gt_bb[:, :, :, :, 3] - gt_bb[:,  :, :, :, 1])
    pred_bb_area = (pred_bb[:, :, :, :, 2] - pred_bb[:, :, :, :, 0]) * \
                   (pred_bb[:, :, :, :, 3] - pred_bb[:, :, :, :, 1])
    union_area = tf.maximum(gt_bb_area + pred_bb_area - intersection_area,
1e-10)
    iou = tf.clip_by_value(intersection_area / union_area, 0.0, 1.0)
    return iou
```

真实值和预测的边界框堆叠在一起。然后在处理负面积的情况下计算面积。边界框坐标不正确时可能会出现负面积。一个框的右侧坐标可能出现在左侧坐标处。由于没有保留边界框的结构，因此负面积必然会发生。计算联合和相交区域，然后进行最终的交并比计算，即计算真实值和预测面积的重叠面积与总面积的比。交并比的计算可以与算法结合来训练目标定位问题。

2. mAP

mAP 用于评估检测算法。mAP 指标是检测到的边界框的精确率和召回率的乘积。mAP 的取值范围是 0 到 100，数字越大越好。mAP 的计算可以通过先分别计算每个类别的平均精确率，然后再计算类别上的平均值来进行。只有当 mAP 高于 0.5 时，检测才被认为是正确的。通过为每个类别绘制精准率/召回曲线，可以将测试图像中的所有检测结果进行组合。曲线下的最终面积可用于算法的比较。mAP 是衡量网络灵敏度的一个很好的衡量指标，同时不会产生很多虚警。我们已经学习了数据集的评估算法，接下来学习目标定位任务的算法。

4.3 目标定位算法

定位算法是第 2 章和第 3 章中学习的材料的扩展。在图像分类中，一幅图像经过了 CNN（卷积神经网络）的若干层。CNN 的最后一层输出了属于每个标签的概率值。这可以扩展到目标定位中。本章将介绍这些思路的运用。

4.3.1　使用滑动窗口定位目标

一种出于直觉的定位方法是使用目标预测一幅图像的几个裁剪部分。可以通过在图像上移动窗口并对每个窗口进行预测来裁剪图像。移动比原图像更小的窗口并根据窗口大小裁剪图像的方法称为**滑动窗口**（sliding window）。对图像的每个裁剪窗口进行预测称为滑动窗口目标检测。

预测可以通过使用紧密裁剪的图像在图像分类问题上训练的深度学习模型来完成。紧密裁剪（close cropping）意味着在整个裁剪图像中只能找到一个目标。滑动窗口的移动必须在图像上保持均匀。图像的每个部分都经过该模型来查找类别。但这种方法有两个问题。

- 它只能找到与窗口大小相同的目标。如果目标大于窗口大小，则滑动窗口会丢失目标。为了克服这一点，我们将使用**尺度空间**（scale space）的概念。
- 另一个问题是，将窗口移到像素上可能会导致错过一些目标。将窗口移到每个像素上会导致大量额外的计算，因此会降低系统速度。为了避免这种情况，我们将在卷积层中引入一个技巧。

接下来介绍一下这两种技术。

1. 尺度空间概念

尺度空间概念是指使用各种尺寸的图像。一幅图像被缩小为较小的尺寸，因此可以使用相同大小的窗口检测较大的目标。可以把图像调整为大小递减的一些尺寸。通过去除替代像素或使用内插来调整图像大小可能会留下一些伪影。因此要迭代地平滑图像和调整图像大小。通过平滑和调整大小获得的图像构成了尺度空间。

在每个单一尺度上滑动的窗口用于目标的定位。运行多个尺度相当于用更大的窗口执行图像。在多个尺度上运行的计算复杂度很高。可以通过快速移动来实现定位，不过要以准确率为代价。这种复杂度使该解决方案无法用于生产环境。为提高效率，滑动窗口的概念可以通过滑动窗口的全卷积实现。

2. 将全连接层作为卷积层进行训练

滑动窗口的问题是计算复杂度。复杂度来自需要针对每个窗口进行预测。已经计算出了重叠区域每个窗口的深度学习特征。在裁剪窗口中重叠区域的这种特征计算可以减少。解决方案是使用全卷积网络，它只需要计算一次特征。为了理解全卷积网络，我们首先看看如何将全连接的层转换为 `convolution_layer`。卷积核更改为相同的大小。滤波器数量与神经元数量相同。它也可以重复用于其他层。更改卷积核的大小是把全连接层转换为 `convolution_layer` 的更简单方法：

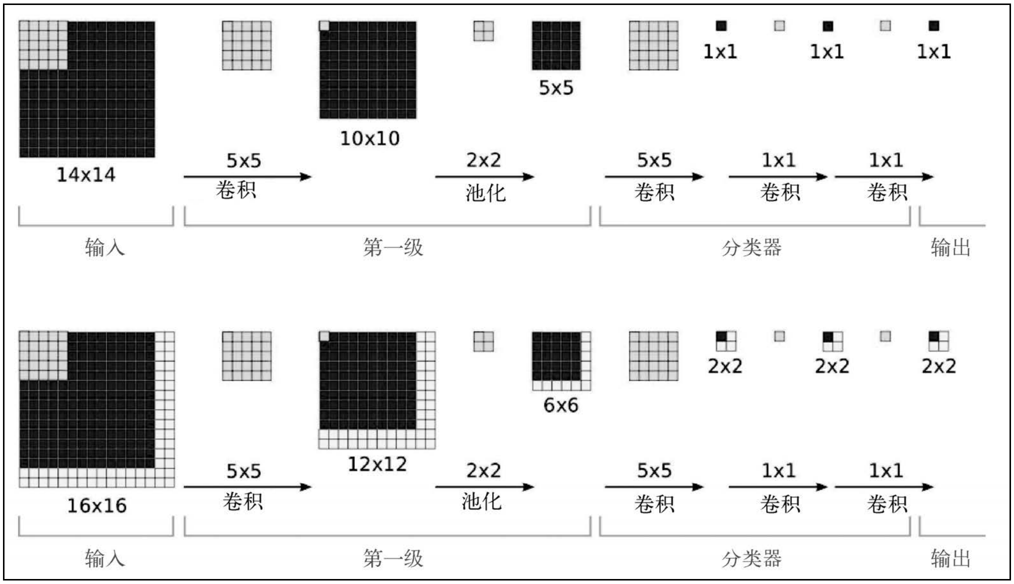
```
convolution_layer_1 = convolution_layer(x_input_reshape, 64)
pooling_layer_1 = pooling_layer(convolution_layer_1)
convolution_layer_2 = convolution_layer(pooling_layer_1, 128)
pooling_layer_2 = pooling_layer(convolution_layer_2)
```

```
dense_layer_bottleneck = convolution_layer(pooling_layer_2, 1024, [5, 5])
logits = convolution_layer(dense_layer_bottleneck, no_classes, [1, 1])
logits = tf.reshape(logits, [-1, 10])
```

密集层表示为卷积层。这个想法在许多情况下是强大和有用的。我们将扩展这个想法，将滑动窗口表示为一个全卷积网络。

3. 滑动窗口的卷积实现

这种技术不使用滑动，而是将最终目标改变为一些要求的目标作为深度并用若干个边界框作为窗口。Sermanet 等人使用全卷积实现来克服滑动窗口的这个问题。图 4-4 是滑动窗口的这种卷积实现的例子。



* 经 Sermanet 等人许可转载

图 4-4

在该图的上半部分，正常分类表示为一个全卷积层。在图的下半部分，相同的卷积核被应用到更大的图像，最后得到 2 × 2 而不是 1 的结果。最后一层表示这些边界框的四个输出。这种容积量的预测可以提高效率，但这些方框仍然存在准确定位的问题。滑动窗口不再是必需的，因此解决了复杂度问题。宽高比（aspect ratio）总是在变化，并且必须在多个尺度上看到。全卷积方法产生的边界框不是很准确。额外的计算仅针对额外的区域完成。可以想象，这些边界框受限于经过训练的边界框的数量。接下来将介绍一种能更准确地检测边界框位置的方法。

4.3.2 将定位看作回归问题

考虑定位的一个基本方法是将问题建模为回归问题。边界框是四个数字，因此可以直接通过

设置回归预测。我们还需要预测标签，这是一个分类问题。

定义边界框有不同的参数化方法。边界框通常有四个数字。其中一种表示是边界高度和宽度的坐标中心。预训练的模型可以通过去掉全连接层并替换为回归编码器来使用。回归需要被 L2 正则化，否则有离群点时会表现不佳。L2 的损失优于 L1。用平滑版本的正则化替换回归更好。对模型进行微调可以提供很好的准确性，而对整个网络进行训练只能得到很小的性能提升。这涉及训练时间和准确性之间的折中。接下来介绍使用卷积网络进行回归的不同应用。

1. 将回归应用于其他问题

回归图像坐标方法适用于其他几种应用，例如姿态检测（pose detection）和基准点检测（fiducial point detection）。如图 4-5 所示，姿态检测是在人体中寻找关节位置的行为。



图 4-5

在前面的图像中，检测到多个位置，例如头部、颈部、肩部、踝部和手部。这可以扩展到所有人体部位。我们学到的回归方法可以用于这个应用。图 4-6 是一个基准点检测的例子。

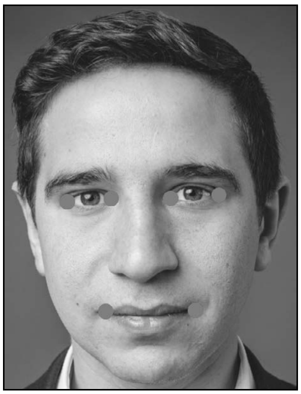


图 4-6

基准点是眼睛、鼻子和嘴唇在脸部相对位置的关键点。寻找这些关键点对于基于人脸的增强现实应用至关重要。脸部还有更多的关键点，这将在第 6 章的人脸识别应用场景下详细介绍。

2. 结合回归与滑动窗口

为了知道该窗口中存在什么目标，滑动窗口方法或全卷积方法中的每个窗口都要计算分类评分。不是为了检测一个目标而预测每个窗口的分类评分，而是可以用分类评分来预测每个窗口本身。结合滑动窗口、尺度空间、全卷积和回归等所有想法，会比任何单独的方法提供更好的结果。图 4-7 是 ImageNet 数据集上使用回归方法通过各种网络实现的前五位定位错误率。

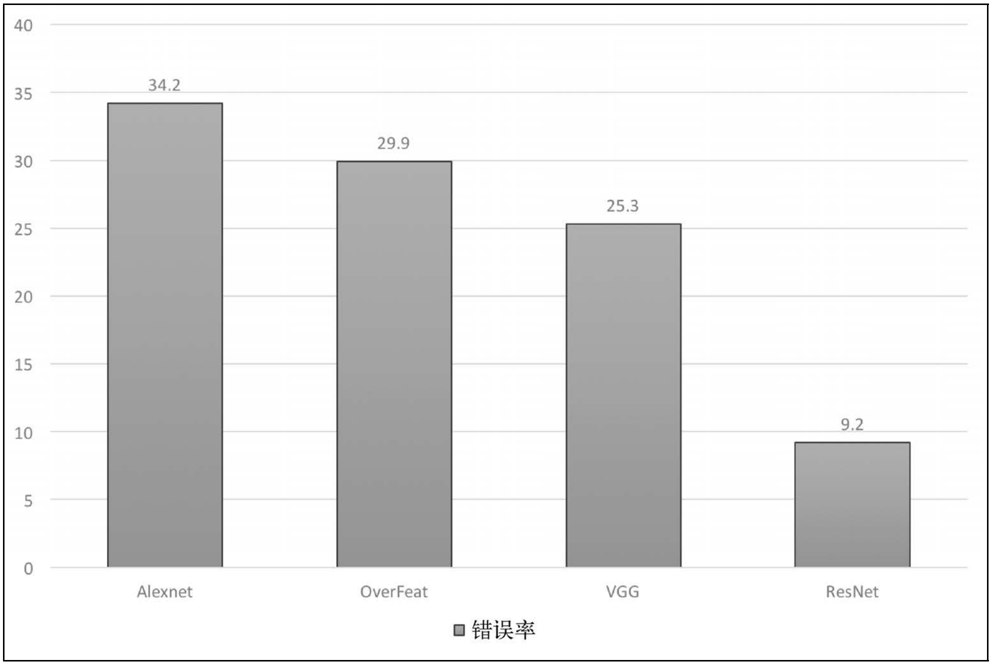


图 4-7

图 4-7 显示：网络越深，结果越好。本文没有描述 AlexNet 的定位方法。OverFeat 使用框合并的多尺度卷积回归。VGG 用了定位，但规模和位置较少。这些增益归因于深层特征。ResNet 使用了不同的定位方法和更深的特征。

回归编码器和分类编码器的功能独立，因此有可能对于一个边界框预测的标签不正确。这个问题可以通过在不同的层上附加回归编码器来解决。该方法也可以用于多个目标，从而解决目标检测问题。给定一幅图像，找到所有的实例。由于输出数量是可变的，因此很难将检测视为回归。一幅图像可能有两个目标，另一幅图像可能有三个或更多目标。在下一节中，我们将学习更有效地处理检测问题的算法。

4.4　检测目标

目标检测算法有几种变体。这里讨论一些目标检测算法和 API。

4.4.1　R-CNN（区域卷积神经网络）

首先是 Girshick 等人提出的区域 CNN。它提出了几个候选框并检查是否有任何框与真实值相符。这些区域建议（region proposal）使用了**选择性搜索**（selective search）。选择性搜索通过对各种尺寸的窗口的颜色/纹理进行分组来提出区域，搜索寻找类似斑点（blob）的结构。它从一个像素开始，并以更高的比例产生一个斑点。它产生了大约 2000 个区域建议。与所有可能的滑动窗口相比，该区域建议较少。

这些建议的大小经过重新调整并通过标准的 CNN 架构，如 Alexnet/VGG/Inception/ResNet。CNN 的最后一层使用 SVM 进行训练，识别具有无目标类中的目标。通过收紧图像周围的边界框，边界得到进一步改进。更近的边界框预测可以用一个线性回归模型和目标区域建议来训练得到。R-CNN 的架构如图 4-8 所示。

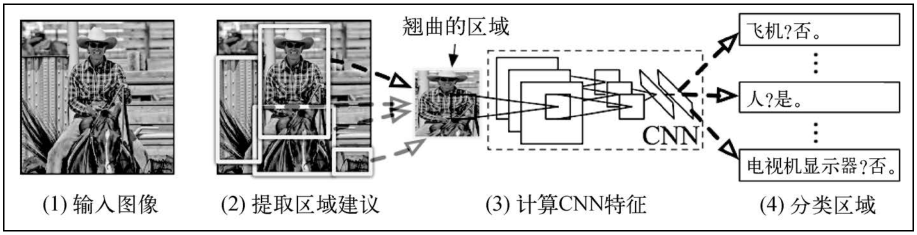


图　4-8

编码器可以是标准深度学习模型的预训练模型。根据训练数据计算所有区域的特征。特征存储下来，然后训练 SVM。接着，用归一化的坐标来训练边界框。在图像坐标之外可能会有一些建议，因此对训练和推断要进行归一化处理。

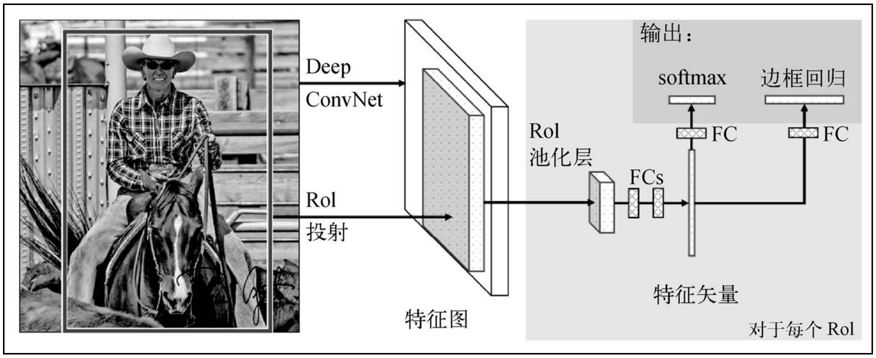
这种方法的缺点是：

- 若干建议是通过选择性搜索形成的，因此必须计算许多推断，数量通常约为 2000；
- 有三个分类器必须进行训练，这会增加参数的数量；
- 没有端到端的训练。

4.4.2　Fast R-CNN

Girshick 等人提出的 Fast R-CNN 方法只运行一次 CNN 推断，从而减少计算量。CNN 的输出用于提出网络并选择边界框。它引入了一种名为感兴趣区域池化（region of interest pooling）的技术。感兴趣区域池化根据区域将 CNN 的特征一起池化。使用 CNN 推断后，池化获得的特征会集

合起来并选择区域，如图 4-9 所示。



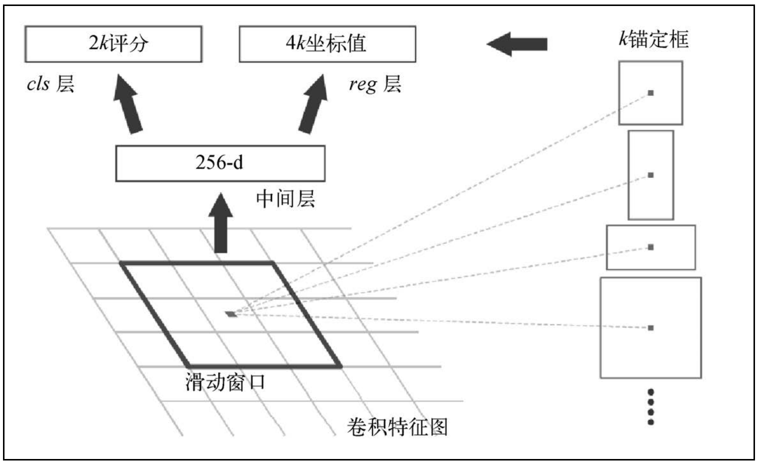
* 经 Girshick 等人许可转载

图 4-9

通过这种方式，可以执行端到端的训练，避免使用多个分类器。请注意，SVM 由 softmax 层替换，框回归（box regressor）由边界框回归（bounding box regressor）取代。这种方法仍然存在选择性搜索的缺点，这需要一定的执行时间。

4.4.3 Faster R-CNN

Faster R-CNN 由 Ren 等人提出。Faster R-CNN 和 Fast R-CNN 方法的区别在于，前者使用 VGG 和 Inception 等架构的 CNN 来代替选择性搜索。CNN 特征进一步经过区域建议网络处理。特征经过一个滑动窗口来输出潜在边界框和评分，也输出一些直观的宽高比。图 4-10 展示了模型输出边界框和评分。



* 经 Ren 等人许可转载

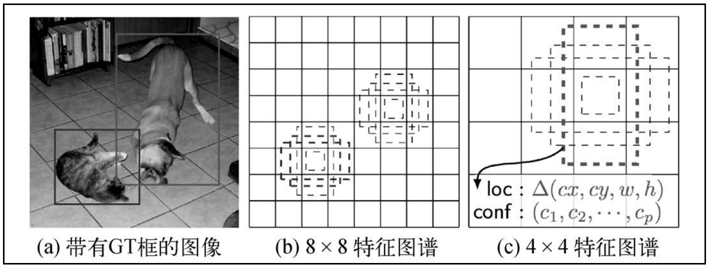
图 4-10

Faster R-CNN 比 Fast R-CNN 更快，因为它仅计算一次特征，这样就节省了计算量。

4.4.4　SSD（单射多框探测器）

SSD（single shot multi-box detector）是所有方法中速度最快的。该方法能够同时预测目标并找到边界框。在训练过程中，可能会有很多负样本，可以使用难分样本挖掘（hard negative mining）来解决类别失衡。CNN 的输出具有各种大小的特征。这些被传递给 3×3 卷积滤波器以预测边界框。

这一步预测目标和边界框，如图 4-11 所示。



* 经 Liu 等人许可转载

图　4-11

这些是可用于目标检测的算法，我们将在下一节中学习如何实现它们。

4.5　目标检测 API

Google 发布了使用各种算法在 COCO 数据集上进行训练得到的预训练模型，以供公众使用。该 API 建立在 TensorFlow 之上，用于构建、训练和部署目标检测模型。这些 API 支持目标检测和定位任务。可用的预训练模型可以对新数据进行微调，从而加快训练速度。这些不同的模型在速度和准确率上各有优缺点。

4.5.1　安装和设置

使用以下命令安装 Protocol Buffers（protobuf）编译器。为 protobuf 创建一个目录并直接下载该库：

```
mkdir protoc_3.3
cd protoc_3.3
wget https://github.com/google/protobuf/releases/download/v3.3.0/protoc-3.
3.0-linux-x86_64.zip
```

更改文件夹的权限并提取内容，如下所示：

```
chmod 775 protoc-3.3.0-linux-x86_64.zip
unzip protoc-3.3.0-linux-x86_64.zip
```

protobuf 用于序列化结构化数据，是 Google 提出的具有语言中立、平台无关和可扩展特点的机制。它提供类似 XML 的服务，但更简单快捷。在 TensorFlow 中，模型通常导出为这种格式。人们只定义一次数据结构，但可以用各种语言读取或写入。然后运行以下命令来编译 protobuf。回到工作文件夹，从 https://github.com/tensorflow/models.git 克隆 repo，并将它们移动到以下文件夹：

```
git clone https://github.com/tensorflow/models.git
```

现在，使用以下代码将模型移动到 research 文件夹：

```
cd models/research/
~/protoc_3.3/bin/protoc object_detection/protos/*.proto --python_out=.
```

TensorFlow 目标检测 API 使用 protobuf 来导出模型权重和训练参数。TensorFlow、models、research 和 slim 目录应通过以下命令附加到 `PYTHONPATH` 中：

```
export PYTHONPATH=.:./slim/
```

4

使用上述命令添加到 python 路径只能运行一次。接下来，这个命令必须再次运行。可以通过运行以下代码来测试安装：

```
python object_detection/builders/model_builder_test.py
```

这段代码的输出如下所示：

```
Ran 7 tests in 0.022s

OK
```

有关安装的更多信息可以从 https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/installation.md 获取。现在已经安装完成并经过了测试。

4.5.2　预训练模型

有几种模型经过预训练是可用的。所有这些模型都在 COCO 数据集上进行了训练，可用于检测 COCO 数据集中存在的目标，例如人和汽车。这些模型对于新任务的迁移学习也很有用，如交通标志检测。表 4-1 显示了一些预训练模型，其中有模型在 COCO 数据集上的相对速度和 mAP。各种不同的算法都是用不同的 CNN 训练的，并用名字来描述。

表　4-1

模型名称	速　　度	COCO mAP
ssd_mobilenet_v1_coco	快	21
ssd_inception_v2_coco	快	24
rfcn_resnet101_coco	中等	30
faster_rcnn_resnet101_coco	中等	32
faster_rcnn_inception_resnet_v2_atrous_coco	慢	37

你可以根据需要从模型中进行选择。下载经过 Mobilenet 训练的 SSD 模型并将其解压缩，进入工作目录，如下所示：

```
mkdir Chapter04 && cd Chapter04
wget
http://download.tensorflow.org/models/object_detection/ssd_mobilenet_v1_coco_11_06
_2017.tar.gz
tar -xzvf ssd_mobilenet_v1_coco_11_06_2017.tar.gz
```

Chapter04 文件夹中包含各种文件，这些文件在此处列出。

- 这是图的原始定义：graph.pbtxt。
- 冻结图的权重，可以用于推断：freeze_inference_graph.pb。
- 检查点文件。
    - model.ckpt.data-00000-of-00001
    - model.ckpt.meta
    - model.ckpt.index

该模型将在下一节中用于检测任务。

4.5.3　重新训练目标检测模型

同样的 API 可以让我们重新训练自定义数据集的模型。训练自定义数据涉及数据集的准备、选择算法和执行微调。整个流水线可以作为一个参数传递给训练脚本。训练数据必须转换为 TensorFlow 记录。TensorFlow 记录是由 Google 提供的一种文件格式，能够比常规文件更快地读取数据。现在，让我们浏览一下训练的步骤。

1. 为宠物数据集做数据准备

本例使用 Oxford-IIIT Pet 数据集。使用 Chapter04 目录中的这些命令下载图像和标定。

```
wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
wget http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
```

提取图像和标定，如下所示：

```
tar -xvf images.tar.gz
tar -xvf annotations.tar.gz
```

创建 pet_tf 记录文件以在 tf 记录中创建数据集，因为它们是目标检测训练器的必需输入。Pet 数据集的 `label_map` 可以在 object_detection/data/pet_label_map.pbtxt 处找到。移至 research 文件夹并运行以下命令：

```
python object_detection/create_pet_tf_record.py \
    --label_map_path=object_detection/data/pet_label_map.pbtxt \
    --data_dir=~/chapter4/. \
    --output_dir=~/chapter4/.
```

可以在名为 pet_train.record 和 pet_val.record 的 research 目录中看到两个.record 文件。

2. 目标检测训练流水线

必须为训练配置好训练 protobuf。以下五件事在这个过程中很重要：

- 模型类型的模型配置；
- 用于标准训练参数的 `train_config`；
- 必须报告的指标 `eval_config`；
- 用于数据集的 `train_input_` 配置；
- 用于评估数据集的 `eval_input_` 配置。

我们将使用 https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssd_mobilenet_v1_pets.config 中的配置文件。通过运行以下命令将其下载到 Chapter04 文件夹。打开配置文件并编辑以下行：

```
fine_tune_checkpoint:
"~/Chapter04/ssd_mobilenet_v1_coco_11_06_2017/model.ckpt"

train_input_reader: {
  tf_record_input_reader {
    input_path: "~/Chapter04/pet_train.record"
  }
  label_map_path:
"~/model/research/object_detection/data/pet_label_map.pbtxt"
}

eval_input_reader: {
  tf_record_input_reader {
    input_path: "~/Chapter04/pet_val.record"
  }
  label_map_path:
"~/model/research/object_detection/data/pet_label_map.pbtxt"
}
```

保存 config 文件。文件中的各种参数会影响模型的准确性。

3. 训练模型

现在 API、数据和配置文件已准备好重新训练。训练可以通过以下命令触发：

```
PYTHONPATH=.:./slim/. python object_detection/train.py \
    --logtostderr \
    --pipeline_config_path=~/chapter4/ssd_mobilenet_v1_pets.config \
    --train_dir=~/Chapter04
```

训练将从约 140 的损失开始，并继续下降。训练会永久运行，必须通过 Ctrl+C 命令手动杀死。训练期间创建的检查点可以稍后用于推断。

4. 使用 TensorBoard 监测损失和准确率

使用 TensorBoard 可以监测训练损失和准确率。使用以下命令运行 TensorBoard：

```
tensorboard --logdir=/home/ubuntu/Chapter04
```

训练和评估都可以在 TensorBoard 中查看。

4.5.4　为自动驾驶汽车训练行人检测

在自动驾驶汽车的情况下，图像上会有四个类用于标记：行人、汽车、摩托车和背景。当没有任何类存在时，必须检测背景类。训练深度学习分类模型的一个假设是，至少有一个目标会出现在图像中。我们通过添加 `background` 类，正努力克服这个问题。神经网络也可以从标签生成目标的边界框。

4.6　YOLO 目标检测算法

You Only Look Once（YOLO）是最近出现的一个目标检测算法。图像分为多个网格，每个网格单元格都运行相同的算法。让我们通过使用初始化器定义一些层来开始实现：

```
def pooling_layer(input_layer, pool_size=[2, 2], strides=2, padding='valid'):
    layer = tf.layers.max_pooling2d(
        inputs=input_layer,
        pool_size=pool_size,
        strides=strides,
        padding=padding
    )
    add_variable_summary(layer, 'pooling')
    return layer

def convolution_layer(input_layer, filters, kernel_size=[3, 3], padding='valid',
                      activation=tf.nn.leaky_relu):
    layer = tf.layers.conv2d(
        inputs=input_layer,
        filters=filters,
        kernel_size=kernel_size,
        activation=activation,
        padding=padding,
        weights_initializer=tf.truncated_normal_initializer(0.0, 0.01),
        weights_regularizer=tf.l2_regularizer(0.0005)
    )
    add_variable_summary(layer, 'convolution')
    return layer

def dense_layer(input_layer, units, activation=tf.nn.leaky_relu):
    layer = tf.layers.dense(
        inputs=input_layer,
        units=units,
        activation=activation,
```

```
        weights_initializer=tf.truncated_normal_initializer(0.0, 0.01),
        weights_regularizer=tf.l2_regularizer(0.0005)
    )
    add_variable_summary(layer, 'dense')
    return layer
```

可以注意到激活层是 leaky_relu，权重是用截断的正态分布初始化的。这些修改后的层可用于构建模型。该模型创建如下：

```
yolo = tf.pad(images, np.array([[0, 0], [3, 3], [3, 3], [0, 0]]), name='pad_1')
yolo = convolution_layer(yolo, 64, 7, 2)
yolo = pooling_layer(yolo, [2, 2], 2, 'same')
yolo = convolution_layer(yolo, 192, 3)
yolo = pooling_layer(yolo, 2, 'same')
yolo = convolution_layer(yolo, 128, 1)
yolo = convolution_layer(yolo, 256, 3)
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = pooling_layer(yolo, 2, 'same')
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = convolution_layer(yolo, 256, 1)
yolo = convolution_layer(yolo, 512, 3)
yolo = convolution_layer(yolo, 512, 1)
yolo = convolution_layer(yolo, 1024, 3)
yolo = pooling_layer(yolo, 2)
yolo = convolution_layer(yolo, 512, 1)
yolo = convolution_layer(yolo, 1024, 3)
yolo = convolution_layer(yolo, 512, 1)
yolo = convolution_layer(yolo, 1024, 3)
yolo = convolution_layer(yolo, 1024, 3)
yolo = tf.pad(yolo, np.array([[0, 0], [1, 1], [1, 1], [0, 0]]))
yolo = convolution_layer(yolo, 1024, 3, 2)
yolo = convolution_layer(yolo, 1024, 3)
yolo = convolution_layer(yolo, 1024, 3)
yolo = tf.transpose(yolo, [0, 3, 1, 2])
yolo = tf.layers.flatten(yolo)
yolo = dense_layer(yolo, 512)
yolo = dense_layer(yolo, 4096)

dropout_bool = tf.placeholder(tf.bool)
yolo = tf.layers.dropout(
        inputs=yolo,
        rate=0.4,
        training=dropout_bool
    )
yolo = dense_layer(yolo, output_size, None)
```

几个卷积层堆叠在一起，产生 YOLO 网络。该网络用于创建实时的目标检测算法。

4.7 小结

在本章中,我们学习了目标定位和目标检测任务之间的区别,讨论了几个数据集和评估标准。还讨论了各种定位问题和算法的方法，例如用于目标检测的 R-CNN 和 SSD 模型的变体，然后又介绍了开放源代码库中检测的实现。我们使用这些技术为行人检测训练了一个模型，还了解了训练此类模型的各种权衡。

在下一章中，我们将学习语义分割算法，并利用这些知识来实现医学成像和卫星图像问题的分割算法。

第 5 章

语义分割

本章中，我们将学习各种语义分割（semantic segmentation）技术，并为此训练模型。分割是一种像素级别的分类任务。解决分割问题的思路是目标检测问题的延伸。分割在医疗图像和卫星图像的理解等应用中非常有用。

本章涉及以下主题：

- 语义分割和实例分割（instance segmentation）之间的区别；
- 分割数据集和指标；
- 语义分割算法；
- 分割在医学图像和卫星图像中的应用；
- 实例分割算法。

5.1 预测像素

图像分类任务预测标签或类别。目标检测任务通过几个深度学习算法预测目标边界框。在边界框内部可能具有除了检测目标之外的其他目标。在一些应用中，将每个像素标记为标签是很重要的，而不只是标记可能具有多个目标的边界框。语义分割是预测像素级标签的任务。

图 5-1 是一幅图像及其相应语义分割的例子。

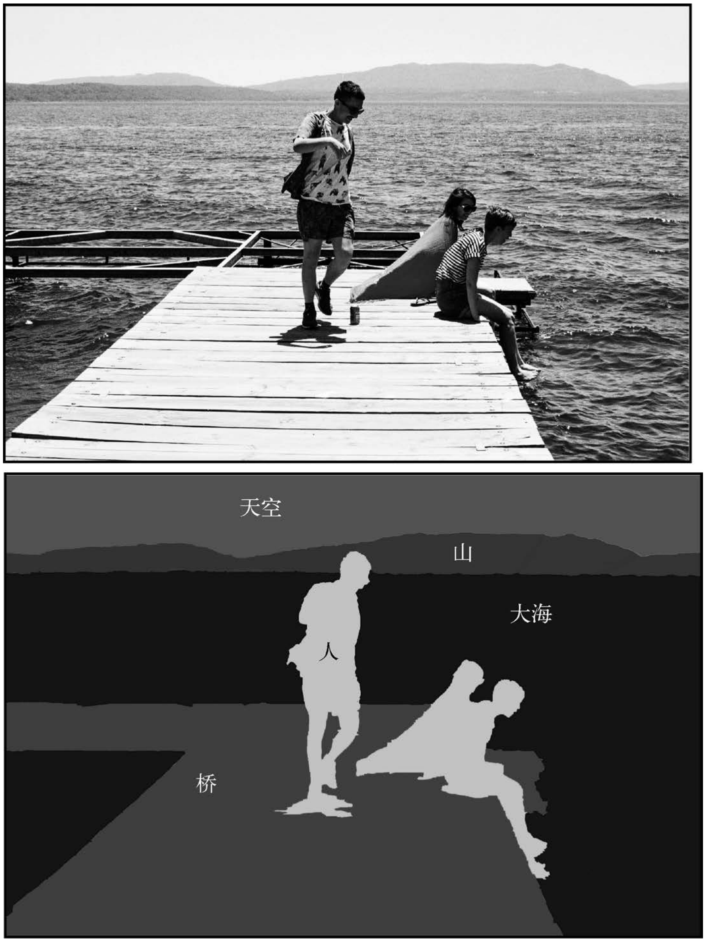


图 5-1

如图 5-1 所示，输入图像对每个像素用标签进行预测。标签可以是天空、大海、人、山和桥。将标签分配给每个像素，而不是分配给整幅图像。语义分割独立地标记像素。你会注意到图像中没有区分任何人，所有人都以相同的方式标记。

图 5-2 中区分了相同标签的实例。



图　5-2

使用像素级标签分割每个实例的任务称为**实例分割**。实例分割可以视为使用像素级标签对目标检测做扩展。语义分割和实例分割的应用非常广泛，在接下来的几节中将介绍其中一些应用。

5.1.1　诊断医学图像

医学图像可以用分割技术来诊断。磁共振成像（MRI）、计算机断层扫描（CT）和视网膜病变等现代医学成像技术能生成高质量的图像。这些技术产生的图像可以分割成不同的区域，以检测脑部扫描的肿瘤或视网膜扫描的斑点。一些设备提供的体积图像也可以通过分割技术进行分析。分割机器人手术视频使医生能够在机器人辅助手术中仔细查看区域。本章后面将介绍如何分割医学图像。

5.1.2　通过卫星图像了解地球

最近卫星图像变得越来越丰富。卫星拍摄的图像提供了地球全表面的高分辨率视图。通过分析卫星图像，我们可以了解关于地球的一些事情，例如：

- 衡量与国家经济增长相关的国家建设速度；
- 测量油库；
- 规划和组织交通；
- 计算森林砍伐的量及其影响；
- 通过对动物进行计数并跟踪它们的移动来帮助保护野生动物；
- 发现考古遗址；
- 绘制受自然灾害影响的受损地区。

卫星图像有更多的应用可能。对于上面提到的大多数问题，解决方案都从卫星图像的分割开始。本章后面将介绍如何分割卫星图像。

5.1.3 提供机器人视觉

分割场景对于机器人看到世界并与周围世界互动至关重要。工业机器人和家庭机器人必须能够处理目标。一旦为机器人提供视觉使其能够跟随目标移动，相应的处理就成为可能。机器人视觉还有一些值得一提的应用。

- 分割缺陷的工业检验工具。
- 时装业的颜色诊断：可以用不同的时尚目标对图像进行分割，并将这些时尚目标用于颜色分析。
- 区分前景和背景以产生肖像效果。

下一节将介绍一些用于评估分割算法的公共数据集。

5.2 数据集

第 4 章中提到的 PASCAL 和 COCO 数据集也可以用于分割任务。但标定是不同的，对于分割任务它们是以像素级标记的。新算法通常以 COCO 数据集为基准。COCO 还具有诸如草、墙和天空之类的数据集。像素准确率属性可以用作评估算法的度量。

为分割任务创建训练数据非常昂贵。可以使用在线工具标定你的数据集，比如麻省理工大学提供的 LabelMe 移动应用程序。

5.3 语义分割算法

为解决图像分割任务，相继提出了几种基于深度学习的算法。滑动窗口方法可以应用于像素级分割。滑动窗口方法获取图像并将图像分解为较小的图块，每个图块都被分类为标签。这种方法既昂贵又低效，因为它不会重用重叠图块之间的共享特征。接下来，我们将讨论一些可以克服这个问题的算法。

5.3.1 全卷积网络

全卷积网络（fully convolutional network，FCN）引入了端到端卷积网络的思想。任何标准 CNN 架构删除全连接层后都可以用于 FCN。在第 4 章中给出了它的实现。全连接层被卷积层取代，最后几层的深度较高，尺寸较小。因此，可以执行 1D 卷积以达到所需的标签数量。但是对于分割，必须保留空间维度。因此，全卷积网络是在没有最大池化的情况下构建的，如图 5-3 所示。

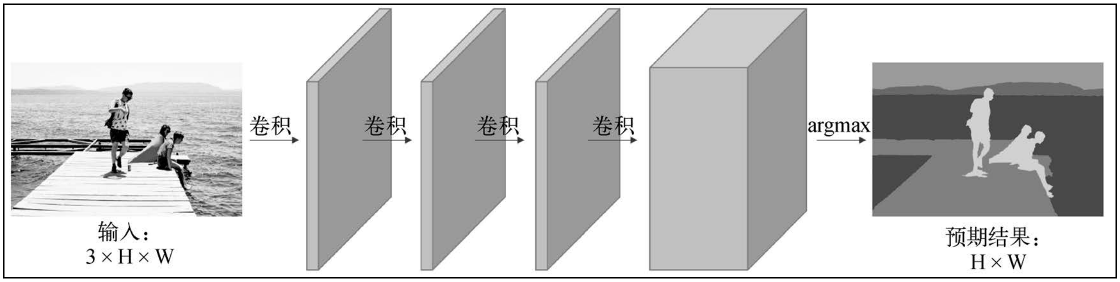


图 5-3

该网络的损失通过平均每个像素和小批次的交叉熵损失来计算。最后一层的深度等于类别的数量。FCN 类似于目标检测，只是空间维度被保留了。由于某些像素可能被错误地预测，因此该架构产生的输出将会很粗糙。FCN 的计算量很高，在下一节中，我们将学习如何解决这个问题。

5.3.2 SegNet 架构

SegNet 具有编码器和解码器方法。它的编码器具有各种卷积层，而解码器具有各种反卷积（deconvolution）层。SegNet 改进了 FCN 的粗糙输出。正因为如此，它只占用较少的内存。当特征维度减小时，通过反卷积再次对图像尺寸进行上采样，从而反转卷积的效果。反卷积学习上采样的参数。由于池化层中的信息丢失，这种架构的输出会很粗糙，如图 5-4 所示。

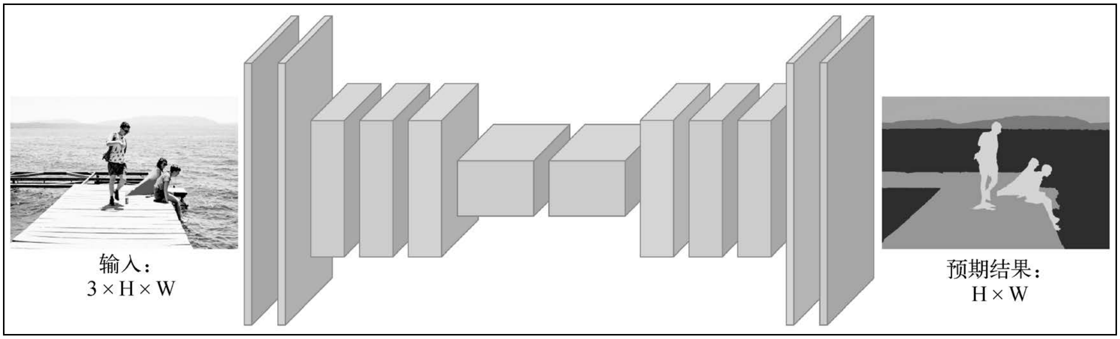


图 5-4

接下来学习一些新的概念：上采样、带洞卷积（atrous convolution）和转置卷积（transpose convolution）。这有助于我们更好地理解这个网络。

1. 通过池化对层进行上采样

在第 1 章中，我们讨论了最大池化。最大池化是一种采样策略，可以从窗口中选取最大值。这对于上采样来说是相反的。每个值都用零包围，从而对层进行上采样，如图 5-5 所示。

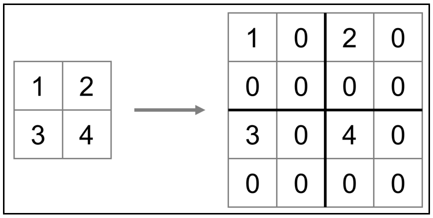


图 5-5

零被添加到与上采样的数字同样的位置。通过记住下采样的位置并将其用于上采样，可以改善去池化（unpooling），如图 5-6 所示。

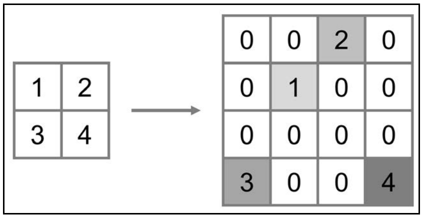


图 5-6

从索引来看，上采样比添加零产生的结果更好。池化对层进行上采样是不通过学习的，而是按规则运行的。接下来介绍如何用可学习参数进行上采样和下采样。

2. 用卷积对层进行采样

可以使用卷积直接对层进行上采样或下采样。增加用于卷积的步幅可以导致下采样，如图 5-7 所示。

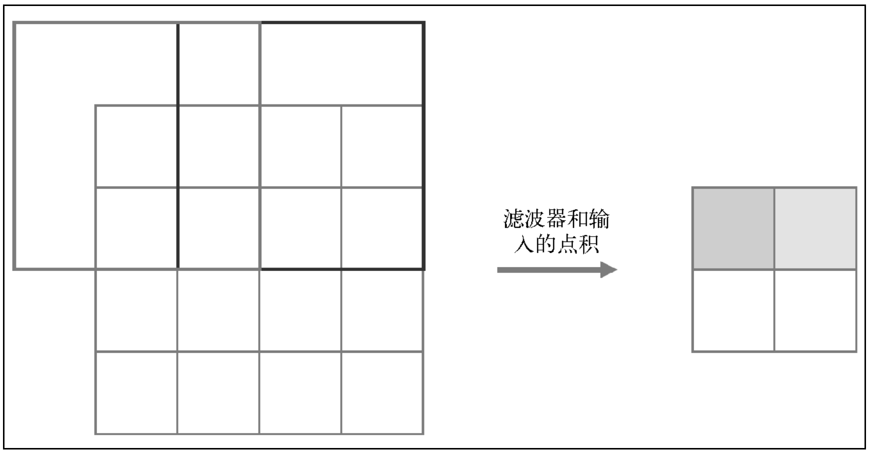


图 5-7

通过卷积进行下采样被称为空洞卷积（atrous convolution）或膨胀卷积（dilated convolution）或步幅卷积（strided convolution）。同样，通过学习如图 5-8 所示的卷积核，它可以反转为上采样。

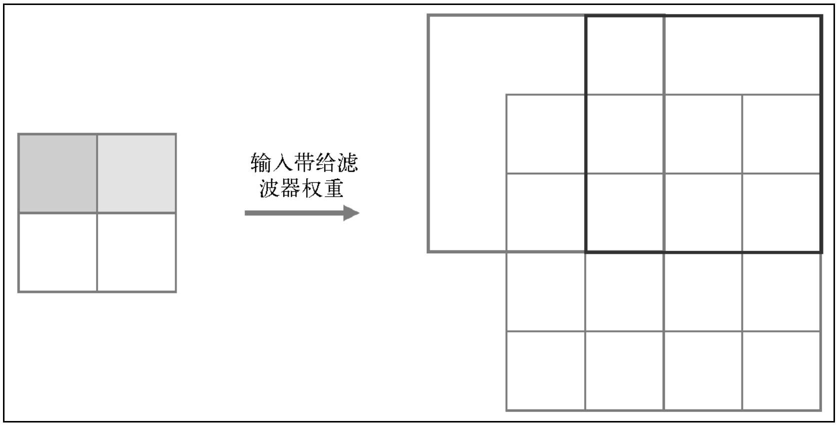


图 5-8

直接使用卷积进行上采样称为转置卷积（transposed convolution）。其同义词有反卷积（deconvolution）、分数步幅卷积（fractionally strided convolution）和上卷积（up-convolution）。现在我们了解了上采样的过程。以下代码描述了之前的算法：

```
input_height = 360
input_width = 480
kernel = 3
filter_size = 64
pad = 1
pool_size = 2
```

接收输入后，接着是尺寸减小的通常的卷积神经网络，可以称为编码器。以下代码可用于定义编码器：

```
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Layer(input_shape=(3, input_height, input_width)))

# encoder
model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(filter_size, kernel, kernel,
                                 border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(pool_size, pool_size)))

model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(128, kernel, kernel, border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation('relu'))
```

```
model.add(tf.keras.layers.MaxPooling2D(pool_size=(pool_size, pool_size)))

model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(256, kernel, kernel, border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation('relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size=(pool_size, pool_size)))

model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(512, kernel, kernel, border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())
model.add(tf.keras.layers.Activation('relu'))
```

编码器的输出可以随着尺寸的增加被送到解码器，如下所示：

```
# decoder
model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(512, kernel, kernel, border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.UpSampling2D(size=(pool_size, pool_size)))
model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(256, kernel, kernel, border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.UpSampling2D(size=(pool_size, pool_size)))
model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(128, kernel, kernel, border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.UpSampling2D(size=(pool_size, pool_size)))
model.add(tf.keras.layers.ZeroPadding2D(padding=(pad, pad)))
model.add(tf.keras.layers.Conv2D(filter_size, kernel, kernel,
border_mode='valid'))
model.add(tf.keras.layers.BatchNormalization())

model.add(tf.keras.layers.Conv2D(nClasses, 1, 1, border_mode='valid', ))
```

解码后的图像与输入尺寸相同，可以使用以下代码训练整个模型：

```
model.outputHeight = model.output_shape[-2]
model.outputWidth = model.output_shape[-1]

model.add(tf.keras.layers.Reshape((nClasses, model.output_shape[-2] *
model.output_shape[-1]),
                  input_shape=(nClasses, model.output_shape[-2],
model.output_shape[-1])))

model.add(tf.keras.layers.Permute((2, 1)))
model.add(tf.keras.layers.Activation('softmax'))

model.compile(loss="categorical_crossentropy",
optimizer=tf.keras.optimizers.Adam, metrics=['accuracy'])
```

这种编码和解码图像的方式克服了基于 FCN 的模型的缺点。接下来将介绍一个使用膨胀卷积的不同概念。

3. 跳跃连接以获得更好的训练

分割输出的粗糙度可以通过跳跃架构（skip architecture）来限制，并且可以获得更高的分辨率。另一种替代方法是按图 5-9 所示放大最后三个层并对它们取平均。

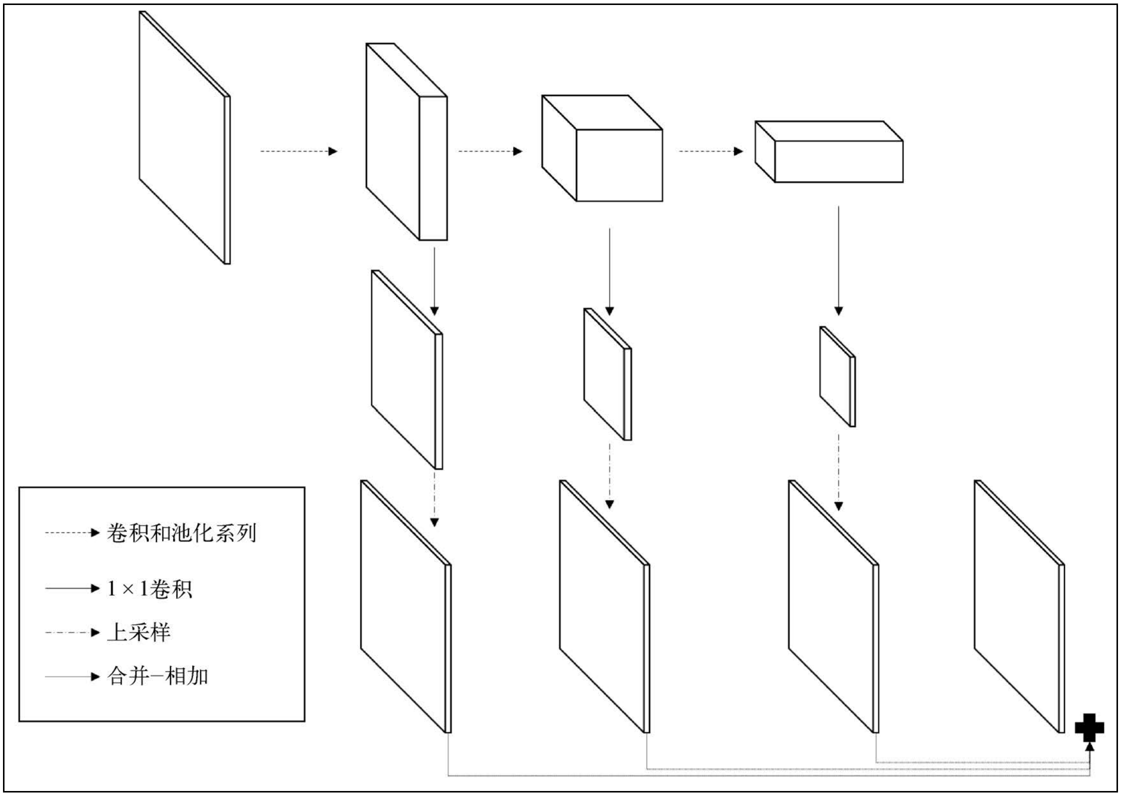


图　5-9

该算法将用于后面的卫星图像示例。

5.3.3　膨胀卷积

像素级的分类和图像分类在结构上是不同的。因此，池化层会减少信息并产生粗糙的分割。但请注意，池化对于获得更广的视野和允许采样至关重要。为了解决这个问题，引入了膨胀卷积。膨胀卷积本质上是通过跳过窗口中的每个像素进行卷积，如图 5-10 所示。

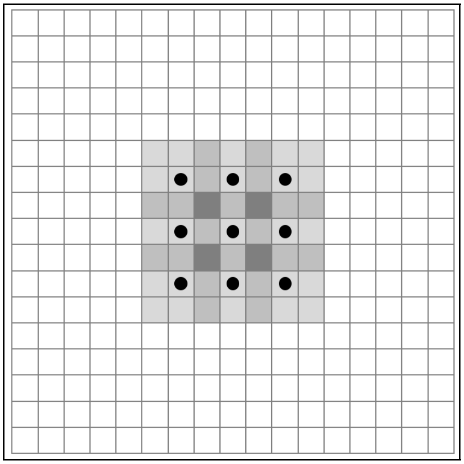
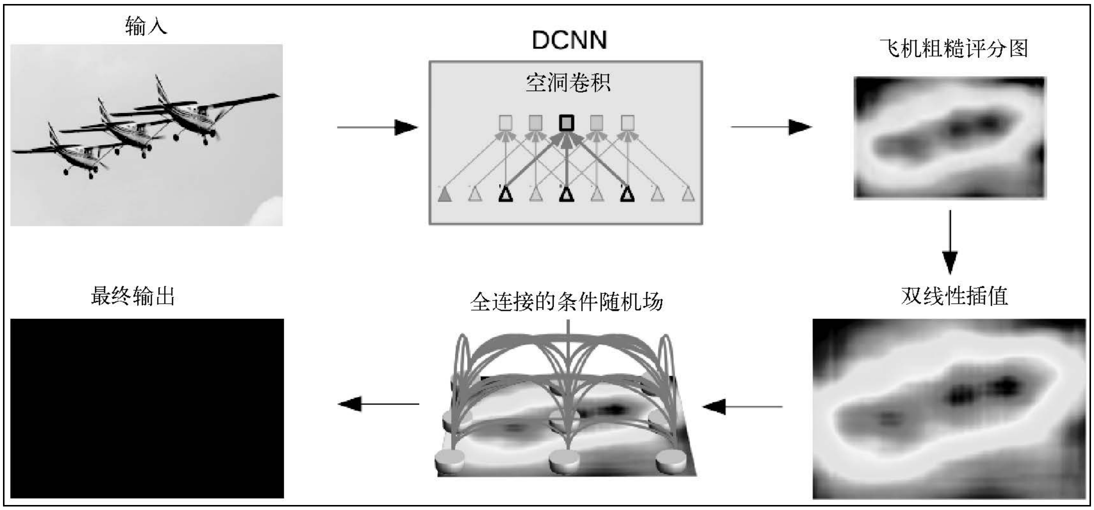


图　5-10

不同层之间的膨胀距离不同。这种分割结果的输出被放大以获得更精细的分辨率。训练一个单独的网络进行多尺度聚合。

5.3.4　DeepLab

Chen 等人提出的 DeepLab 在多个尺度上执行卷积，并使用各种尺度的特征来获得评分图。然后插入评分图，通过**条件随机场**（CRF）进行最终分割。各种尺寸的图像可以用 CNN 处理，或者通过不同级别的膨胀卷积的并行卷积来实现，如图 5-11 所示。

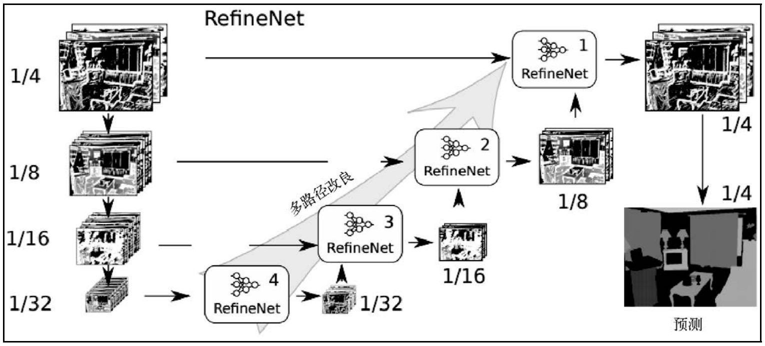


* 经 Chen 等人许可转载

图　5-11

5.3.5 RefiNet

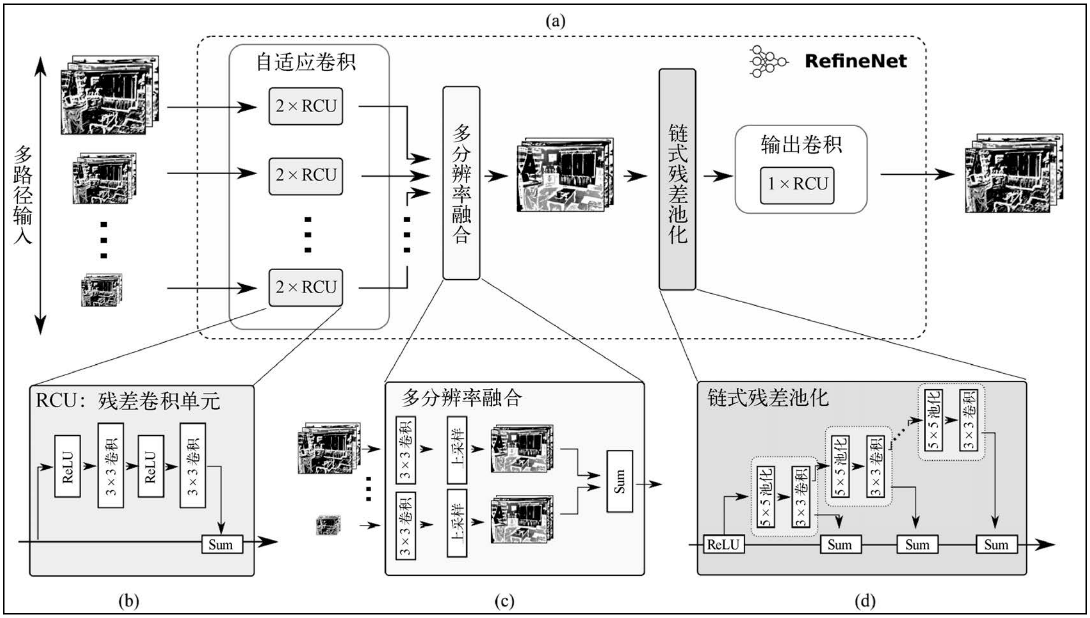
膨胀卷积需要更大的输入量，因此内存需求大。在使用高分辨率图像时，这会呈现计算问题。Reid 等人提出了一种名为 RefiNet 的方法来应对这个问题，如图 5-12 所示。



* 经 Reid 等人许可转载

图 5-12

RefiNet 使用一个编码器，然后是一个解码器。CNN 的编码器输出，解码器连接各种大小的特征，如图 5-13 所示。



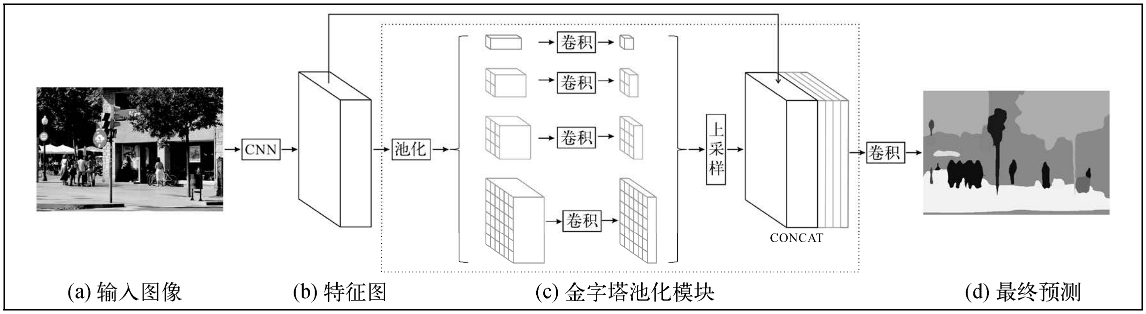
* 经 Reid 等人许可转载

图 5-13

通过对低维度特征的放大，完成了拼接。

5.3.6　PSPnet

Zhao 等人引入的 PSPnet 使用全局内容，通过增加池化层的核大小来实现（图 5-14）。池化以金字塔形式进行。金字塔同时涵盖了图像的各个部分和大小。在能够进行适度监督的架构之间存在一定的损失。

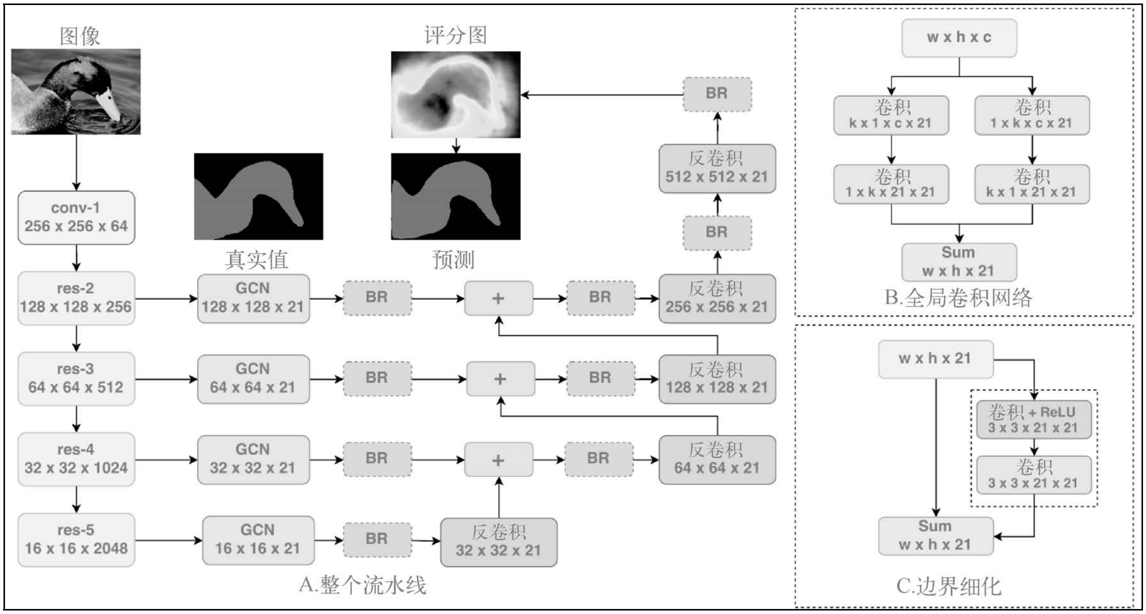


* 经 Zhao 等人许可转载

图　5-14

5.3.7　大卷积核的重要性

Peng 等人展示了大卷积核的重要性（图 5-15）。大核拥有比小核更大的感受野。这些大核的计算复杂性可以用一个近似的较小的核来克服。最后给出一个边界细化网络。



* 经 Peng 等人许可转载

图　5-15

5.3.8 DeepLab v3

Chen 等人提出了批归一化，用以提高性能。特征的多尺度以级联方式编码，如图 5-16 和图 5-17 所示。

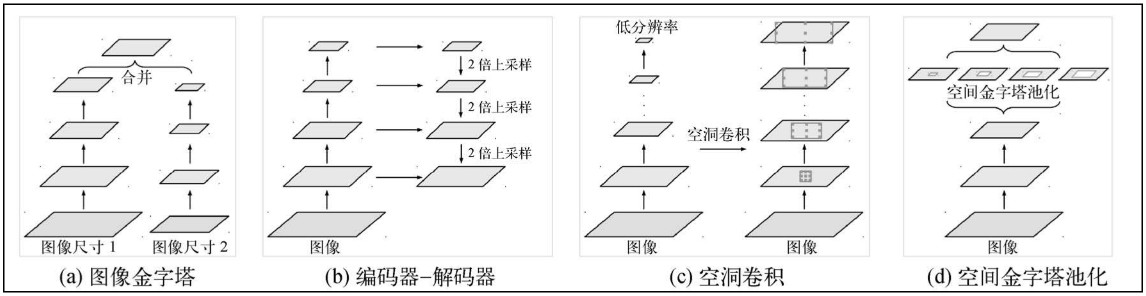


图 5-16

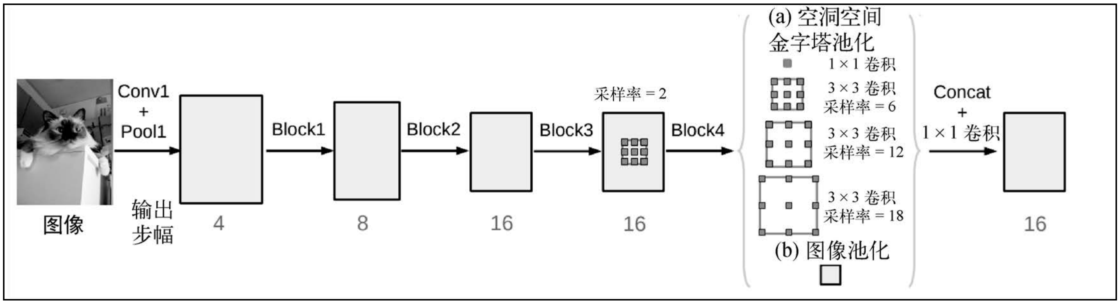
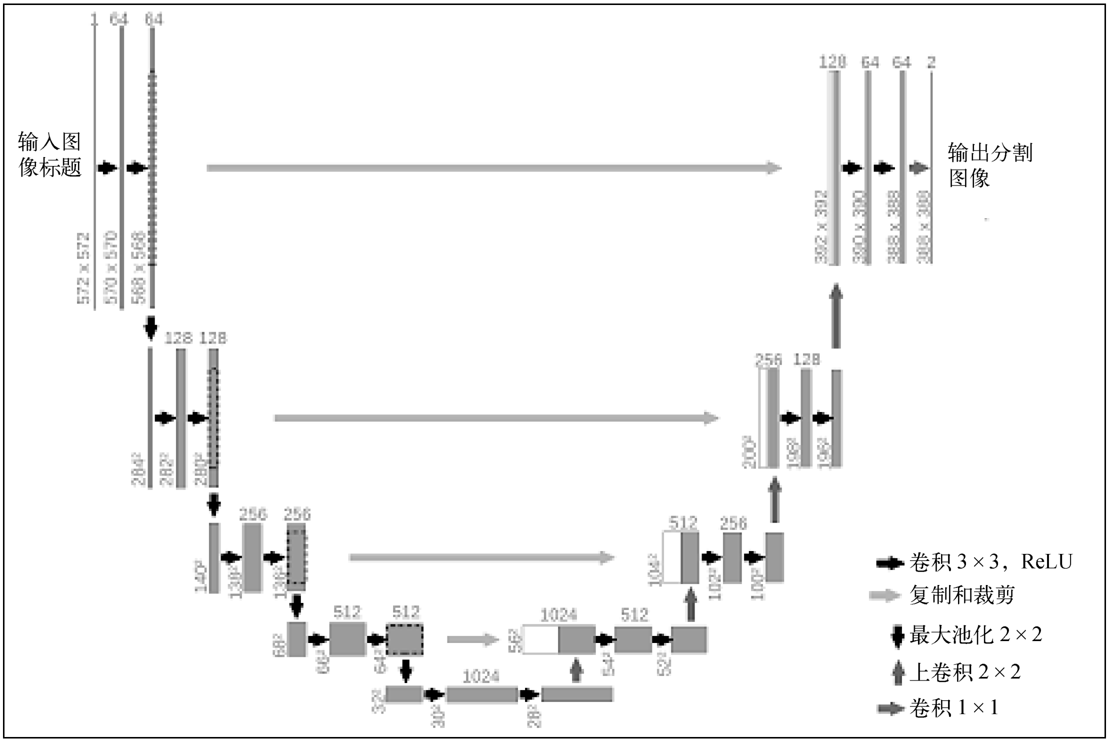


图 5-17

上述几种架构使用深度学习提高了图像分割的准确率。接下来，我们将学习它在医疗成像的应用。

5.4 超神经分割

Kaggler 是一个在预测建模和分析方面进行竞赛的组织，曾组织挑战从颈部超声波图像中分割神经结构。Ronneberger 等人提出的 UNET 模型类似于一个自编码器，但使用的是卷积而不是全连接层。该模型有一个编码器和一个解码器。编码器部分使用卷积减少维度，解码器部分增加维度，如图 5-18 所示。



* 经 Ronneberger 等人许可转载

图　5-18

类似大小的编码器和解码器部分的卷积通过层间的跳跃连接（skip connection）进行学习。模型的输出是一个范围在 0 到 1 之间的掩码。让我们从导入函数开始，如下所示：

```
import os
from skimage.transform import resize
from skimage.io import imsave
import numpy as np
from data import load_train_data, load_test_data
```

在所有导入之后，使用以下代码定义尺寸：

```
image_height, image_width = 96, 96
smoothness = 1.0
work_dir = ''
```

现在定义 dice_coefficient 及其损失函数。在该例子中，dice_coefficient 也是度量指标：

```
def dice_coefficient(y1, y2):
    y1 = tf.flatten(y1)
    y2 = tf.flatten(y2)
```

```
    return (2. * tf.sum(y1 * y2) + smoothness) / (tf.sum(y1) + tf.sum(y2) + smoothness)

def dice_coefficient_loss(y1, y2):
    return -dice_coefficient(y1, y2)
```

UNET 模型可以如下定义：

```
def preprocess(imgs):
    imgs_p = np.ndarray((imgs.shape[0], image_height, image_width),
dtype=np.uint8)
    for i in range(imgs.shape[0]):
        imgs_p[i] = resize(imgs[i], (image_width, image_height),
preserve_range=True)
    imgs_p = imgs_p[..., np.newaxis]
    return imgs_p


def covolution_layer(filters, kernel=(3,3), activation='relu',
input_shape=None):
    if input_shape is None:
        return tf.keras.layers.Conv2D(
            filters=filters,
            kernel=kernel,
            activation=activation)
    else:
        return tf.keras.layers.Conv2D(
            filters=filters,
            kernel=kernel,
            activation=activation,
            input_shape=input_shape)


def concatenated_de_convolution_layer(filters):
    return tf.keras.layers.concatenate([
        tf.keras.layers.Conv2DTranspose(
            filters=filters,
            kernel=(2, 2),
            strides=(2, 2),
            padding='same'
    )],
    axis=3
)
```

所有层都被拼接并使用，如下所示：

```
unet = tf.keras.models.Sequential()
inputs = tf.keras.layers.Input((image_height, image_width, 1))
input_shape = (image_height, image_width, 1)
unet.add(covolution_layer(32, input_shape=input_shape))
unet.add(covolution_layer(32))
unet.add(pooling_layer())

unet.add(covolution_layer(64))
unet.add(covolution_layer(64))
unet.add(pooling_layer())
```

```
unet.add(covolution_layer(128))
unet.add(covolution_layer(128))
unet.add(pooling_layer())

unet.add(covolution_layer(256))
unet.add(covolution_layer(256))
unet.add(pooling_layer())

unet.add(covolution_layer(512))
unet.add(covolution_layer(512))
```

层被拼接起来，并且使用反卷积层：

```
unet.add(concatenated_de_convolution_layer(256))
unet.add(covolution_layer(256))
unet.add(covolution_layer(256))

unet.add(concatenated_de_convolution_layer(128))
unet.add(covolution_layer(128))
unet.add(covolution_layer(128))

unet.add(concatenated_de_convolution_layer(64))
unet.add(covolution_layer(64))
unet.add(covolution_layer(64))

unet.add(concatenated_de_convolution_layer(32))
unet.add(covolution_layer(32))
unet.add(covolution_layer(32))

unet.add(covolution_layer(1, kernel=(1, 1), activation='sigmoid'))
unet.compile(optimizer=tf.keras.optimizers.Adam(lr=1e-5),
             loss=dice_coefficient_loss,
             metrics=[dice_coefficient])
```

接下来，可以用图像训练模型，如下所示：

```
x_train, y_train_mask = load_train_data()

x_train = preprocess(x_train)
y_train_mask = preprocess(y_train_mask)

x_train = x_train.astype('float32')
mean = np.mean(x_train)
std = np.std(x_train)

x_train -= mean
x_train /= std

y_train_mask = y_train_mask.astype('float32')
y_train_mask /= 255.

unet.fit(x_train, y_train_mask, batch_size=32, epochs=20, verbose=1,
shuffle=True,
```

```
            validation_split=0.2)

x_test, y_test_mask = load_test_data()
x_test = preprocess(x_test)

x_test = x_test.astype('float32')
x_test -= mean
x_test /= std

y_test_pred = unet.predict(x_test, verbose=1)

for image, image_id in zip(y_test_pred, y_test_mask):
    image = (image[:, :, 0] * 255.).astype(np.uint8)
    imsave(os.path.join(work_dir, str(image_id) + '.png'), image)
```

图像可以被预处理和使用。现在可以进行图像的训练和测试了。模型训练好后，分割会产生不错的结果，如图 5-19 所示。

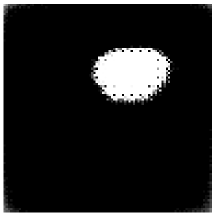


图 5-19

我们已经训练了一种可以分割医学图像的模型。该算法可以用于多种用例。在下一节中，我们将学习分割卫星图像。

5.5 分割卫星图像

在本节中，我们将使用国际摄影测量和遥感学会（International Society for Photogrammetry and Remote Sensing，ISPRS）提供的数据集。该数据集包含德国波茨坦卫星图像，分辨率为 5 厘米。这些图像附带图像的红外数据和高度轮廓数据。有六个与图像相关的标签：

- 建筑物；
- 植被；
- 树；
- 出租车；
- 杂波（clutter）；
- 抗渗（impervious）。

该数据集共提供 38 幅图像和 6000 × 6000 个图块。请转到页面填写表单，在表单上选择以下选项（图 5-20）。

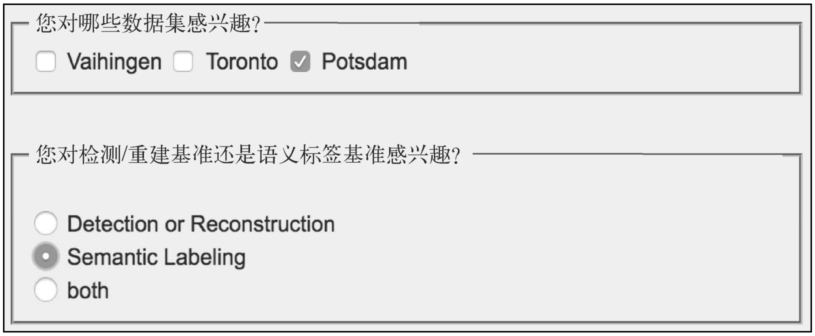


图　5-20

发送表单后，会发送给你一封电子邮件，可以从中下载数据。

为分割建模 FCN

导入库并获取输入的形状。标签数量定义为 6：

```
from .resnet50 import ResNet50
nb_labels = 6

img_height, img_width, _ = input_shape
input_tensor = tf.keras.layers.Input(shape=input_shape)
weights = 'imagenet'
```

在 ImageNet 上预训练过的 ResNet 模型将被用作基础模型。使用 ResNet 定义基础模型，如下所示：

```
resnet50_model = ResNet50(
    include_top=False, weights='imagenet', input_tensor=input_tensor)
```

使用以下代码从 ResNet 中获取最后三个层：

```
final_32 = resnet50_model.get_layer('final_32').output
final_16 = resnet50_model.get_layer('final_16').output
final_x8 = resnet50_model.get_layer('final_x8').output
```

每个跳跃连接（skip connection）都必须进行压缩以匹配等于标签数的通道：

```
c32 = tf.keras.layers.Conv2D(nb_labels, (1, 1))(final_32)
c16 = tf.keras.layers.Conv2D(nb_labels, (1, 1))(final_16)
c8 = tf.keras.layers.Conv2D(nb_labels, (1, 1))(final_x8)
```

可以使用双线性插值来调整压缩跳跃连接的输出大小。插值可以通过计算 TensorFlow 操作的 Lambda 层实现。使用 lambda 层进行插值，如下所示：

```
def resize_bilinear(images):
    return tf.image.resize_bilinear(images, [img_height, img_width])
```

```
r32 = tf.keras.layers.Lambda(resize_bilinear)(c32)
r16 = tf.keras.layers.Lambda(resize_bilinear)(c16)
r8 = tf.keras.layers.Lambda(resize_bilinear)(c8)
```

我们定义的三个层可以通过添加三个值进行合并，使用以下代码：

```
m = tf.keras.layers.Add()([r32, r16, r8])
```

可以使用 softmax 激活函数来应用模型的概率。该模型在应用 softmax 前后调整大小：

```
x = tf.keras.ayers.Reshape((img_height * img_width, nb_labels))(m)
x = tf.keras.layers.Activation('img_height')(x)
x = tf.keras.layers.Reshape((img_height, img_width, nb_labels))(x)

fcn_model = tf.keras.models.Model(input=input_tensor, output=x)
```

一个简单的 FCN 层已经被定义，在训练后会得出如图 5-21 所示的结果。

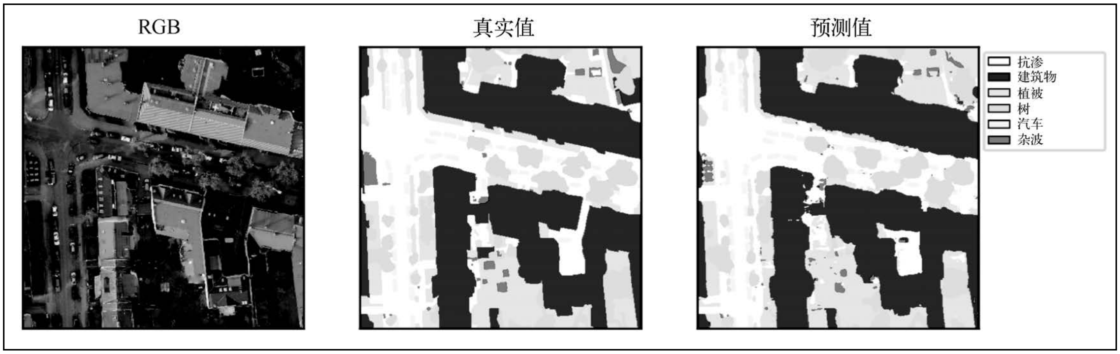
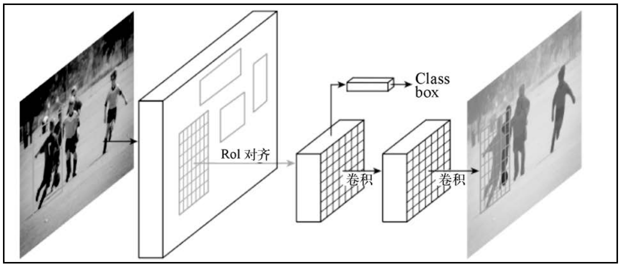


图　5-21

你可以看到六个标签的预测是合理的。接下来学习分割实例。

5.6　分割实例

在分析图像时，我们的兴趣只会被图像中的某些实例吸引。因此，不得不从图像的其余部分分割出这些实例。将所需信息与其他信息分离的过程泛称为分割实例。在此过程中，首先输入图像，然后对边界框与目标进行定位，最后为每个类预测一个像素级的掩码。计算每个目标的像素级准确度。有几种分割实例的算法，最近的一种算法是由 He 等人提出的 Mask RCNN 算法。图 5-22 描绘了 Mask R-CNN 的架构。



* 经 He 等人许可转载

图　5-22

该体系架构看起来与 R-CNN 类似，只是增加了分割。这是一个多级的网络，提供端到端的训练。它的区域建议是习得的。该网络分为两部分，一部分用于检测，另一部分用于分类评分。分割结果非常好，对象被准确检测并相应地进行分割，如图 5-23 所示。



* 经 He 等人许可转载

图　5-23

同一个网络也可以预测人的姿态。分割和检测任务是并行处理的。

5.7　小结

在本章中，我们了解了各种分割算法，看到了用于基准测试的数据集和度量指标，应用所学的技术来分割卫星图像和医学图像，讨论了针对实例分割的 Mask R-CNN 算法。

在下一章中，我们将学习相似性学习以及可用于相似性学习的模型架构。相似性学习模型用于学习两幅图像之间的比较机制，适用于人脸识别等应用。

第 6 章 相似性学习

在本章中，我们将学习相似性学习，以及其中使用的各种损失函数。当每个类的数据集较小时，相似性学习会很有用。我们将了解可用于人脸分析的不同数据集，并构建人脸识别和特征点检测的模型。

本章涉及以下主题：

- 相似性学习的不同算法；
- 用于相似性学习的各种损失函数；
- 可以使用这些模型的各种场景；
- 人脸识别的完整过程。

6.1 相似性学习算法

相似性学习是训练一个度量指标以计算两个实体之间相似程度的过程。这也可以被称为度量学习（metric learning），因为相似度是习得的。度量可以是欧几里得距离、余弦距离或其他自定义距离函数。实体可以是任何数据，如图像、视频、文本或表格。计算度量需要图像的向量表示。这种表示可以是由 CNN 计算的特征，如第 3 章所述。习得目标分类的 CNN 可以用作计算度量的向量。用于图像分类获得的特征向量不是目前应用的最佳表示。在相似性学习中，我们找到了产生用于相似性学习任务训练特征的 CNN。这里给出相似性学习的一些应用：

- 验证人脸的生物特征，比较两张脸；
- 对现实世界的物体进行可视化搜索，以查找类似的线上产品；
- 对某些属性类似产品的视觉推荐。

本章将详细介绍人脸验证。让我们从可用于相似性学习的算法开始。

6.1.1 孪生网络

顾名思义，孪生网络（siamese network）就是为区分两个输入所训练的神经网络模型。一个孪生网络可以通过两个编码器训练一个 CNN 而产生一个嵌入（embedding）。每个编码器都送入

一幅从一个正对或一个负对中提供的图像。孪生网络比其他深度学习算法需要的数据更少。最初引入孪生网络是为了比较签名。图 6-1 显示了一个孪生网络，权重在网络之间共享。

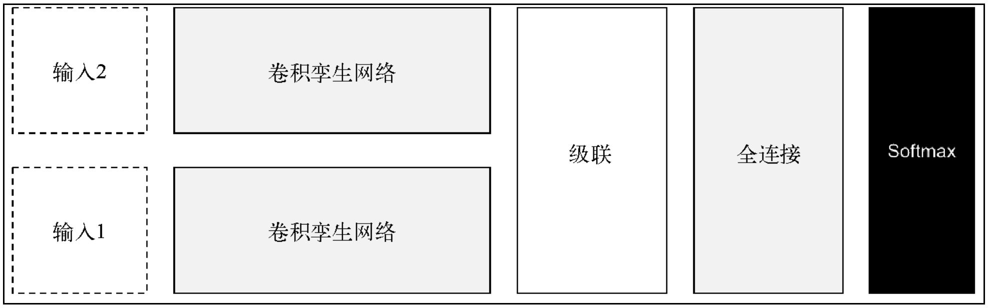


图　6-1

孪生网络的另一个用途是**单样本学习**（one-short learning）。单样本学习就是学习只有一个样本的技巧。在这种情况下，只显示一幅图像，就可以判断它和样本是否相似。对于大多数相似性学习任务来说，需要一对正负对来训练。假设这些数据集是欧几里得距离，则可以使用用于分类任务的任何数据集来形成这些数据集。这些算法和前几章算法的主要区别在于：编码器试图把它们一个个地区分开。

对比损失

对比损失（contrastive　loss）通过相似性区分图像。使用相似性指标来比较特征或潜在层，并用一个相似度评分的目标进行训练。在正对的情况下，相应的目标将为 0，因为两个输入是相同的。对于负对，在余弦距离或正则化欧几里得距离的情况下，潜在对之间的距离最大为 0。损失可以通过 `contrastive_loss` 来定义，如下所示：

```
def contrastive_loss(model_1, model_2, label, margin=0.1):
    distance = tf.reduce_sum(tf.square(model_1 - model_2), 1)
    loss = label * tf.square(
        tf.maximum(0., margin - tf.sqrt(distance))) + (1 - label) *
distance
    loss = 0.5 * tf.reduce_mean(loss)
    return loss
```

比较两个模型的距离并计算损失。现在，我们将定义和训练一个孪生网络。对于孪生网络，我们需要两个相同的模型。接下来，让我们为具有给定输入的简单 CNN 定义一个函数，如下所示：

```
def get_model(input_):
    input_reshape = tf.reshape(input_, [-1, 28, 28, 1],
                               name='input_reshape')
    convolution_layer_1 = convolution_layer(input_reshape, 64)
    pooling_layer_1 = pooling_layer(convolution_layer_1)
    convolution_layer_2 = convolution_layer(pooling_layer_1, 128)
    pooling_layer_2 = pooling_layer(convolution_layer_2)
```

```
    flattened_pool = tf.reshape(pooling_layer_2, [-1, 5 * 5 * 128],
                                name='flattened_pool')
    dense_layer_bottleneck = dense_layer(flattened_pool, 1024)
    return dense_layer_bottleneck
```

定义的模型将被使用两次，以定义孪生网络所需的编码器。接下来，定义两个模型的占位符。每一对输入的相似性也作为输入送入。定义的模型是相同的。模型也可以定义为权重共享。这里定义了左侧和右侧的两个模型：

```
left_input = tf.placeholder(tf.float32, shape=[None, input_size])
right_input = tf.placeholder(tf.float32, shape=[None, input_size])
y_input = tf.placeholder(tf.float32, shape=[None, no_classes])
left_bottleneck = get_model(left_input)
right_bottleneck = get_model(right_input)
```

瓶颈层取自模型并级联起来。这对相似性学习问题至关重要。可以创建任意数量的模型，最后的几层可以级联起来，如下所示：

```
dense_layer_bottleneck = tf.concat([left_bottleneck, right_bottleneck], 1)
```

接下来，添加一个丢弃层，并在级联层中计算出 logits。该过程与其他任何网络都是类似的，如下所示：

```
dropout_bool = tf.placeholder(tf.bool)
dropout_layer = tf.layers.dropout(
        inputs=dense_layer_bottleneck,
        rate=0.4,
        training=dropout_bool
    )
logits = dense_layer(dropout_layer, no_classes)

with tf.name_scope('loss'):
    softmax_cross_entropy = tf.nn.softmax_cross_entropy_with_logits(
        labels=y_input, logits=logits)
    loss_operation = tf.reduce_mean(softmax_cross_entropy, name='loss')
    tf.summary.scalar('loss', loss_operation)

with tf.name_scope('optimiser'):
    optimiser = tf.train.AdamOptimizer().minimize(loss_operation)

with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        predictions = tf.argmax(logits, 1)
        correct_predictions = tf.equal(predictions, tf.argmax(y_input, 1))
    with tf.name_scope('accuracy'):
        accuracy_operation = tf.reduce_mean(
            tf.cast(correct_predictions, tf.float32))
tf.summary.scalar('accuracy', accuracy_operation)

session = tf.Session()
session.run(tf.global_variables_initializer())

merged_summary_operation = tf.summary.merge_all()
train_summary_writer = tf.summary.FileWriter('/tmp/train', session.graph)
```

```
test_summary_writer = tf.summary.FileWriter('/tmp/test')

test_images, test_labels = mnist_data.test.images, mnist_data.test.labels
```

数据必须单独送到左右两个模型，如下所示：

```
for batch_no in range(total_batches):
    mnist_batch = mnist_data.train.next_batch(batch_size)
    train_images, train_labels = mnist_batch[0], mnist_batch[1]
    _, merged_summary = session.run([optimiser, merged_summary_operation],
                                    feed_dict={
        left_input: train_images,
        right_input: train_images,
        y_input: train_labels,
        dropout_bool: True
    })
    train_summary_writer.add_summary(merged_summary, batch_no)
    if batch_no % 10 == 0:
        merged_summary, _ = session.run([merged_summary_operation,
                                        accuracy_operation], feed_dict={
        left_input: test_images,
        right_input: test_images,
        y_input: test_labels,
        dropout_bool: False
    })
    test_summary_writer.add_summary(merged_summary, batch_no)
```

我们已经学习了如何定义一个孪生网络。定义两个编码器，并将这些隐空间级联起来以形成训练损失。左侧和右侧模型分别获得数据馈送。接下来将介绍如何在单个网络中执行相似性学习。

6.1.2　FaceNet 模型

Schroff 等人提出了 FaceNet 模型来解决人脸验证问题。它学习一个深度 CNN，然后将人脸图像转换为嵌入（embedding）。嵌入可用于比较不同人脸的相似度，具体可以通过以下三种方式使用。

- **人脸验证**（face verification）：考量两个脸部，决定它们是否相似。可以通过计算距离度量来完成。
- **人脸识别**（face recognition）：用名称标注人脸的分类问题。嵌入向量可用于训练最终标签。
- **人脸聚类**（face clustering）：将类似的人脸聚合在一起，就像照片应用程序将同一个人的照片聚集起来一样。k 均值等聚类算法可用于对人脸进行聚类。

图 6-2 显示了 FaceNet 架构。



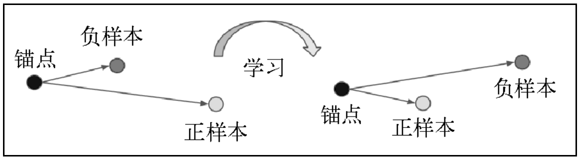
* 经 Schroff 等人许可转载

图　6-2

FaceNet 需要一批脸部图像并训练它们。批次中会有一些正对。在计算损失时，考虑正对和最接近的几个负对。挖掘选择性配对可实现平滑训练。如果所有的负样本一直被排除在外，那么训练并不稳定。比较三个数据点称为三元组损失（triplet loss）。在计算损失时，图像的一个正样本对匹配一个负样本对。负样本对只会以一定的范围被推开。这里详细解释一下三元组损失。

三元组损失

三元组损失学习图像的评分向量。人脸描述子的评分向量可以用来在欧几里得空间中验证人脸。在学习投影的意义上，三元组损失可以区分输入，这与度量学习相似。这些投影、描述子或评分向量是一种简洁的表示形式，因此可以被视为一种降维技术。三元组包括锚点、正样本和负样本。锚点可以是任何样本，正样本是同一个人的图像，负样本图像可能来自另一个人。显而易见，对于一个给定的锚点会有很多负样本。通过选择当前靠近锚点的负样本，编码器难以辨别人脸，从而使其更好地学习。这个过程被称为**难分样本挖掘**（hard negative mining）。相近的负样本可以通过设置一个欧几里得空间的阈值来获得。图 6-3 显示了三元组损失模型。



* 经 Schroff 等人许可转载

图 6-3

TensorFlow 中的损失计算如下所示：

```
def triplet_loss(anchor_face, positive_face, negative_face, margin):
    def get_distance(x, y):
        return tf.reduce_sum(tf.square(tf.subtract(x, y)), 1)

    positive_distance = get_distance(anchor_face, positive_face)
    negative_distance = get_distance(anchor_face, negative_face)
    total_distance = tf.add(tf.subtract(positive_distance,
negative_distance), margin)
    return tf.reduce_mean(tf.maximum(total_distance, 0.0), 0)
```

三元组的挖掘是一项困难的任务。每个点都必须与其他点进行比较才能获得正确的锚点和正对。这里显示了三元组的挖掘过程：

```
def mine_triplets(anchor, targets, negative_samples):
    distances = cdist(anchor, targets, 'cosine')
    distances = cdist(anchor, targets, 'cosine').tolist()
    QnQ_duplicated = [
        [target_index for target_index, dist in enumerate(QnQ_dist) if dist
== QnQ_dist[query_index]]
        for query_index, QnQ_dist in enumerate(distances)]
    for i, QnT_dist in enumerate(QnT_dists):
        for j in QnQ_duplicated[i]:
```

```
            QnT_dist.itemset(j, np.inf)

    QnT_dists_topk = QnT_dists.argsort(axis=1)[:, :negative_samples]
    top_k_index = np.array([np.insert(QnT_dist, 0, i) for i, QnT_dist in
enumerate(QnT_dists_topk)])
    return top_k_index
```

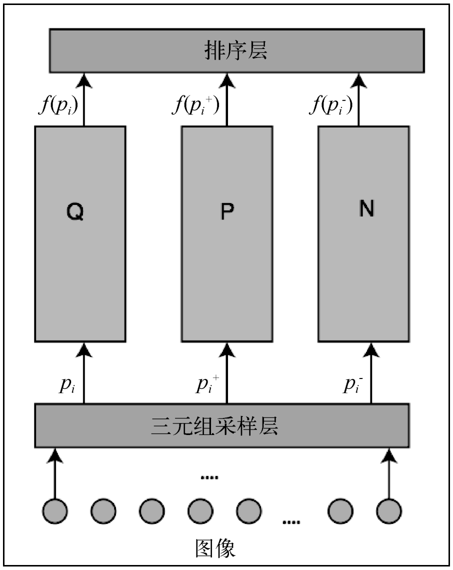
因为距离计算发生在 CPU 中，这可能会使 GPU 机器上的训练速度变慢。FaceNet 模型是一种训练人脸相似度模型的先进的方法。

6.1.3　DeepNet 模型

DeepNet 模型用于学习人脸验证任务的面部嵌入，如 FaceNet。这改进了上一节讨论的 FaceNet 方法。DeepNet 需要同一张脸的多个裁剪，并通过多个编码器获得更好的嵌入。这实现了比 FaceNet 更好的准确性，但需要更多时间进行处理。面部裁剪在同一区域制作并通过各自的编码器。然后将所有层级联起来进行训练，以对抗三元组损失。

6.1.4　DeepRank 模型

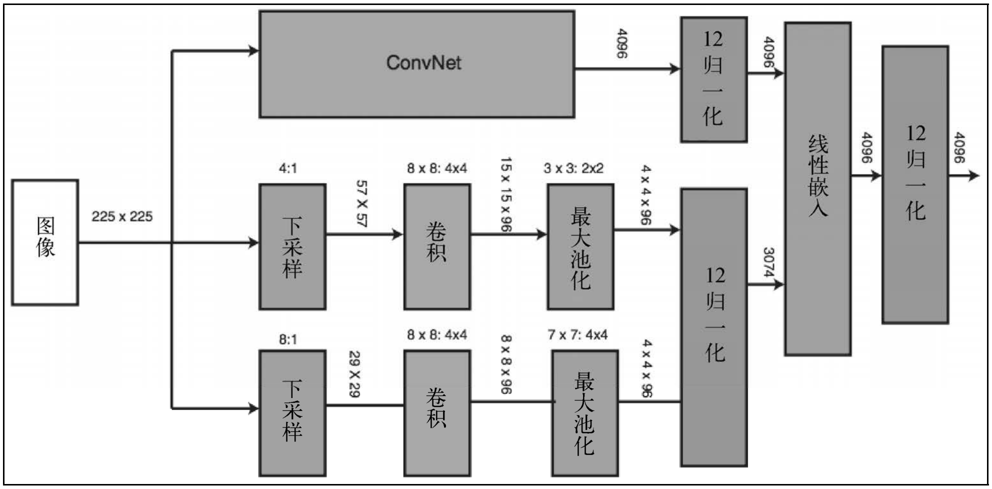
Wang 等人提出了 DeepRank 模型，用于图像的相似性排序。图像通过不同的模型传递，如图 6-4 所示。



* 经 Wang 等人许可转载

图　6-4

这里也计算了三元组损失，反向传播进行得更顺利。然后，图像可以转换为线性嵌入进行排序，如图 6-5 所示。



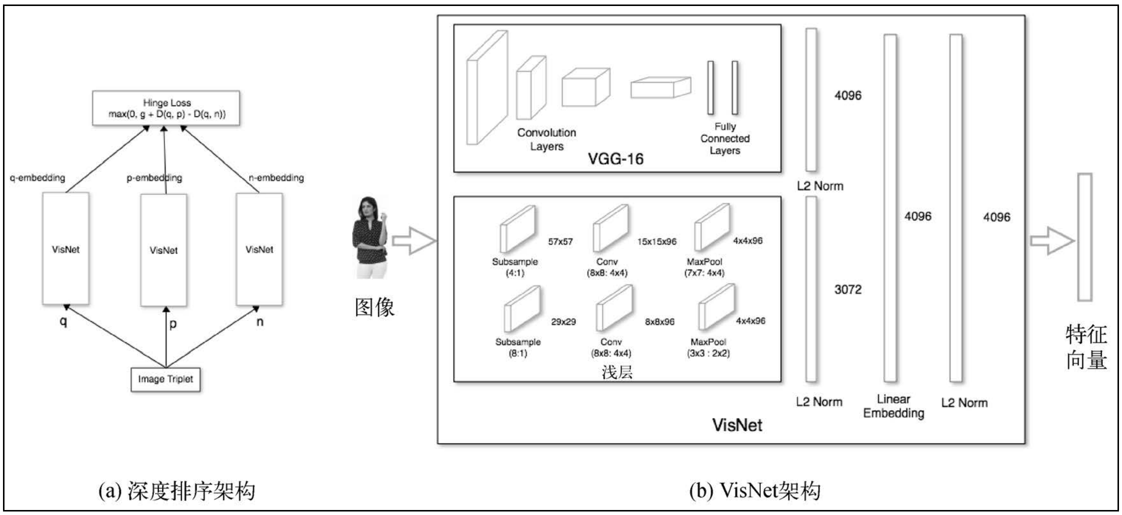
* 经 Wang 等人许可转载

图　6-5

该算法对于排序非常有用。

6.1.5　视觉推荐系统

视觉推荐系统非常适合获取一个给定图像的推荐。推荐模型提供具有相似属性的图像。在 Shankar 等人提出的模型（图 6-6）中，可以学习相似图像的嵌入，而且它还提供了视觉推荐。



* 经 Shankar 等人许可转载

图　6-6

这些是用于相似性学习的一些算法。在下一节中，我们会学习如何将这些技术应用于人脸。

6.2　人脸分析

可以使用计算机视觉以多种方式分析人脸。这里列出几个需要考虑的因素。

- 人脸检测：找到人脸位置的边界框。
- 人脸关键点检测：查找人脸特征的空间点，如鼻子、嘴巴等。
- 人脸对齐：将人脸转换为正面进行进一步分析。
- 属性识别：查找属性，如性别、笑容等。
- 情绪分析：分析人的情绪。
- 人脸验证：查找两幅图像是否属于同一个人。
- 人脸识别：找到人脸的身份。
- 人脸聚类：将同一个人的面部聚类。

在下面的章节中，我们将详细了解这些任务的数据集和实现。

6.2.1　人脸检测

人脸检测与目标检测（参见第 4 章）类似。必须从图像中检测人脸的位置。FDDB（face detection data set and benchmark）数据集包含 2845 幅图像，5171 张人脸。另一个名为 wider face 的数据集包含 32 203 幅图像和 393 703 张人脸。图 6-7 是来自 wider face 数据集的图像示例。



* 经 Yang 等人许可转载

图　6-7

该数据集中人脸的尺寸、姿态、遮挡、表情、化妆和照明都有显著的变化。另一个名为多属性标记的人脸（multi-attribute labelled face，MALF）的数据集包含 5250 幅图像和 11 931 张人脸。目标检测中的技术同样也可以应用于人脸检测。

6.2.2 人脸特征点和属性

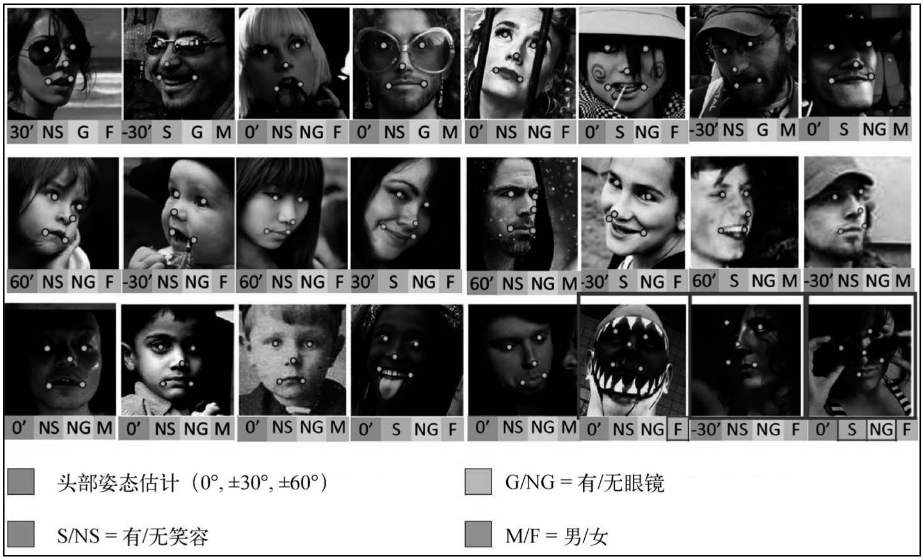
人脸特征点（face landmark）是人脸上的某些空间点。这些空间点对应各种面部特征的位置，例如眼睛、眉毛、鼻子、嘴巴和下巴。根据标定的不同，点数可能从 5 到 78 个不等。人脸特征点也被称为基准点，面部关键点或人脸姿态。人脸特征点的应用如下：

- 对齐脸部以获得更好的人脸验证或人脸识别；
- 追踪视频中的人脸；
- 测量面部表情或情绪；
- 有助于诊断身体状况。

接下来介绍一些具有基准点标定的数据库。

1. 多任务面部特征点（MTFL）数据集

MTFL 数据集由 Zhang 等人提出。该数据集注明了 5 个面部特征点以及性别、笑容、有无眼镜和头部姿态标定。数据库中有 12 995 张人脸。图 6-8 是 MTFL 中的图像示例。



* 经 Zhang 等人许可转载

图 6-8

在年龄、照明、情绪等方面，脸上有很多变化。头部姿态是人脸的朝向，用角度表示；有无眼镜、笑容、性别属性等都使用二进制标签进行注释。

2. Kaggle 关键点数据集

Kaggle 关键点数据集标定了 15 个面部特征点。数据集中有 8832 幅图像，图像大小为 96 像素 × 96 像素。

3. 多属性面部特征点（MAFL）数据集

MAFL 数据集由 Zhang 等人提出。该数据集标定了 5 个面部特征点和 40 个不同的面部属性。数据库中有 2 万张人脸。图 6-9 是 MAFL 中的图像示例。



* 经 Liu 等人许可转载

图　6-9

标定的属性包括尖鼻子、刘海、胡须、波浪发、戴帽子等。这些图像也包含在 CelebA 数据集中，稍后将对其进行详细讨论。

4. 学习面部关键点

正如前面所讨论的，在计算面部关键点时有几个参数需要定义，如下所示：

```
image_size = 40
no_landmark = 10
no_gender_classes = 2
no_smile_classes = 2
no_glasses_classes = 2
no_headpose_classes = 5
batch_size = 100
total_batches = 300
```

接下来，为各种输入提供几个占位符：

```
image_input = tf.placeholder(tf.float32, shape=[None, image_size,
image_size])
landmark_input = tf.placeholder(tf.float32, shape=[None, no_landmark])
gender_input = tf.placeholder(tf.float32, shape=[None, no_gender_classes])
smile_input = tf.placeholder(tf.float32, shape=[None, no_smile_classes])
glasses_input = tf.placeholder(tf.float32, shape=[None,
no_glasses_classes])
headpose_input = tf.placeholder(tf.float32, shape=[None,
no_headpose_classes])
```

接着用四个卷积层构造主模型，如下所示：

```
image_input_reshape = tf.reshape(image_input, [-1, image_size, image_size, 1],
                             name='input_reshape')

convolution_layer_1 = convolution_layer(image_input_reshape, 16)
pooling_layer_1 = pooling_layer(convolution_layer_1)
convolution_layer_2 = convolution_layer(pooling_layer_1, 48)
pooling_layer_2 = pooling_layer(convolution_layer_2)
convolution_layer_3 = convolution_layer(pooling_layer_2, 64)
pooling_layer_3 = pooling_layer(convolution_layer_3)
convolution_layer_4 = convolution_layer(pooling_layer_3, 64)
flattened_pool = tf.reshape(convolution_layer_4, [-1, 5 * 5 * 64],
                            name='flattened_pool')
dense_layer_bottleneck = dense_layer(flattened_pool, 1024)
dropout_bool = tf.placeholder(tf.bool)
dropout_layer = tf.layers.dropout(
        inputs=dense_layer_bottleneck,
        rate=0.4,
        training=dropout_bool
    )
```

接下来为所有不同的任务创建一个 `logits` 分支，如下所示：

```
landmark_logits = dense_layer(dropout_layer, 10)
smile_logits = dense_layer(dropout_layer, 2)
glass_logits = dense_layer(dropout_layer, 2)
gender_logits = dense_layer(dropout_layer, 2)
headpose_logits = dense_layer(dropout_layer, 5)
```

单独为所有面部特征计算损失，如下所示：

```
landmark_loss = 0.5 * tf.reduce_mean(
    tf.square(landmark_input, landmark_logits))

gender_loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        labels=gender_input, logits=gender_logits))

smile_loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        labels=smile_input, logits=smile_logits))
```

```
glass_loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        labels=glasses_input, logits=glass_logits))

headpose_loss = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(
        labels=headpose_input, logits=headpose_logits))

loss_operation = landmark_loss + gender_loss +
                 smile_loss + glass_loss + headpose_loss
```

现在，我们将初始化优化器并开始训练，如下所示：

```
optimiser = tf.train.AdamOptimizer().minimize(loss_operation)
session = tf.Session()
session.run(tf.initialize_all_variables())
fiducial_test_data = fiducial_data.test

for batch_no in range(total_batches):
    fiducial_data_batch = fiducial_data.train.next_batch(batch_size)
    loss, _landmark_loss, _ = session.run(
        [loss_operation, landmark_loss, optimiser],
        feed_dict={
        image_input: fiducial_data_batch.images,
        landmark_input: fiducial_data_batch.landmarks,
        gender_input: fiducial_data_batch.gender,
        smile_input: fiducial_data_batch.smile,
        glasses_input: fiducial_data_batch.glasses,
        headpose_input: fiducial_data_batch.pose,
        dropout_bool: True
})
if batch_no % 10 == 0:
    loss, _landmark_loss, _ = session.run(
        [loss_operation, landmark_loss],
        feed_dict={
            image_input: fiducial_test_data.images,
            landmark_input: fiducial_test_data.landmarks,
            gender_input: fiducial_test_data.gender,
            smile_input: fiducial_test_data.smile,
            glasses_input: fiducial_test_data.glasses,
            headpose_input: fiducial_test_data.pose,
            dropout_bool: False
        })
```

这个过程可以用来检测面部特征以及关键点。

6.2.3　人脸识别

人脸识别（face recognition）或面部识别（facial recognition）是从数字图像或视频中识别特定的人的过程。我们将在下面的章节中了解可用于人脸识别的数据集。

1. 野外标记的人脸（LFW）数据集

LFW 数据集包含 13 233 幅人脸图像，其中包含 5749 个独立的人，被视为评估人脸验证的标准数据集。可以用准确率度量指标来评估算法。

2. YouTube 人脸数据集

YouTube 人脸数据集包含 3425 个视频片段，其中包含 1595 个独立的人。这些视频来自 YouTube，数据集中每人至少有两个视频。该数据集被视为视频中的人脸验证的标准数据集。

3. CelebFaces 属性数据集（CelebA）

CelebA 数据集标定了人的身份以及 5 个面部特征点和 40 个属性。数据库中有 10 177 个独立的人，202 599 幅人脸图像。它是一个大型数据集，可用于人脸验证、检测、特征点和属性识别问题。这些图像具有不同人脸和不同注释。

4. CASIA 网络人脸数据库

CASIA 数据集总共标定了 10 575 个独立的人和 494 414 幅图像。这是用于人脸验证和人脸识别问题的第二大公共数据集。

5. VGGFace2 数据集

Cao 等人提出的 VGGFace2 数据集标注有 9131 个独立的人，313 万幅图像。图像标定了不同的年龄、种族、姿态，职业和照明。这是可用于人脸验证的最大数据集。

图 6-10 是数据集中图像的示例。



* 经 Cao 等人许可转载

图　6-10

每个独立的人的最小、平均和最大图像数量分别为 87 幅、362.6 幅和 843 幅。

6. 计算人脸之间的相似度

人脸相似度的计算是一个多步骤的问题。必须检测人脸，然后找到基准点。人脸可以用基准点对齐，对齐的人脸就可以用于比较了。正如前面提到的，人脸检测类似于目标检测。因此，为了找到人脸之间的相似度，我们将首先导入所需的库和 facenet 库，如下所示：

```
from scipy import misc
import tensorflow as tf
import numpy as np
import os
import facenet
print facenet
from facenet import load_model, prewhiten
import align.detect_face
```

图像的加载和对齐如下所示：

```
def load_and_align_data(image_paths, image_size=160, margin=44,
gpu_memory_fraction=1.0):
    minsize = 20
    threshold = [0.6, 0.7, 0.7]
    factor = 0.709

    print('Creating networks and loading parameters')
    with tf.Graph().as_default():
        gpu_options =
tf.GPUOptions(per_process_gpu_memory_fraction=gpu_memory_fraction)
        sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options,
log_device_placement=False))
        with sess.as_default():
            pnet, rnet, onet = align.detect_face.create_mtcnn(sess, None)

    nrof_samples = len(image_paths)
    img_list = [None] * nrof_samples
    for i in range(nrof_samples):
        img = misc.imread(os.path.expanduser(image_paths[i]), mode='RGB')
        img_size = np.asarray(img.shape)[0:2]
        bounding_boxes, _ = align.detect_face.detect_face(img, minsize,
pnet, rnet, onet, threshold, factor)
        det = np.squeeze(bounding_boxes[0, 0:4])
        bb = np.zeros(4, dtype=np.int32)
        bb[0] = np.maximum(det[0] - margin / 2, 0)
        bb[1] = np.maximum(det[1] - margin / 2, 0)
        bb[2] = np.minimum(det[2] + margin / 2, img_size[1])
        bb[3] = np.minimum(det[3] + margin / 2, img_size[0])
        cropped = img[bb[1]:bb[3], bb[0]:bb[2], :]
        aligned = misc.imresize(cropped, (image_size, image_size),
interp='bilinear')
        prewhitened = prewhiten(aligned)
        img_list[i] = prewhitened
    images = np.stack(img_list)
    return images
```

接下来处理图像路径以获取嵌入。这里给出了相同的代码：

```
def get_face_embeddings(image_paths, model='/20170512-110547/'):
    images = load_and_align_data(image_paths)
    with tf.Graph().as_default():
            with tf.Session() as sess:
                load_model(model)
                images_placeholder =
tf.get_default_graph().get_tensor_by_name("input:0")
                embeddings =
tf.get_default_graph().get_tensor_by_name("embeddings:0")
                phase_train_placeholder =
tf.get_default_graph().get_tensor_by_name("phase_train:0")
                feed_dict = {images_placeholder: images,
phase_train_placeholder: False}
                emb = sess.run(embeddings, feed_dict=feed_dict)

    return emb
```

现在我们将使用以下代码计算嵌入之间的距离：

```
def compute_distance(embedding_1, embedding_2):
    dist = np.sqrt(np.sum(np.square(np.subtract(embedding_1,
embedding_2))))
    return dist
```

该函数将计算嵌入之间的欧几里得距离。

7. 找到最佳的阈值

使用上述函数，可以计算该系统的准确度。以下代码可用于计算最佳阈值：

```
import sys
import argparse
import os
import re
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
```

现在，使用以下代码从文件夹中获取图像路径：

```
def get_image_paths(image_directory):
    image_names = sorted(os.listdir(image_directory))
    image_paths = [os.path.join(image_directory, image_name) for image_name
in image_names]
    return image_paths
```

图像的距离是在传递嵌入时获得的，如下所示：

```
def get_labels_distances(image_paths, embeddings):
    target_labels, distances = [], []
    for image_path_1, embedding_1 in zip(image_paths, embeddings):
        for image_path_2, embedding_2 in zip(image_paths, embeddings):
            if (re.sub(r'\d+', '', image_path_1)).lower() ==
```

```
(re.sub(r'\d+', '', image_path_2)).lower():
                target_labels.append(1)
            else:
                target_labels.append(0)
            distances.append(compute_distance(embedding_1, embedding_2)) #
Replace distance metric here
    return target_labels, distances
```

阈值变化如下所示，并且相应地打印各种指标：

```
def print_metrics(target_labels, distances):
    accuracies = []
    for threshold in range(50, 150, 1):
        threshold = threshold/100.
        predicted_labels = [1 if dist <= threshold else 0 for dist in
distances]
        print("Threshold", threshold)
        print(classification_report(target_labels, predicted_labels,
target_names=['Different', 'Same']))
        accuracy = accuracy_score(target_labels, predicted_labels)
        print('Accuracy: ', accuracy)
        accuracies.append(accuracy)
    print(max(accuracies))
```

现在，借助以下代码将图像路径传递给嵌入：

```
def main(args):
    image_paths = get_image_paths(args.image_directory)
    embeddings = get_face_embeddings(image_paths) # Replace your embedding
calculation here
    target_labels, distances = get_labels_distances(image_paths,
embeddings)
    print_metrics(target_labels, distances)
```

最后，图像的目录作为主要参数传递给这些方法，如下所示：

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('image_directory', type=str, help='Directory
containing the images to be compared')
    parsed_arguments = parser.parse_args(sys.argv[1:])
    main(parsed_arguments)
```

在这个示例中，我们采用了一个预训练模型，并用它来构造一个人脸验证方法。

6.2.4　人脸聚类

人脸聚类是将相同人物的图像分组在一起以进行相册处理的过程。可以提取面部的嵌入，并使用诸如 k 均值聚类这样的算法将同一个人的面部聚集在一起。TensorFlow 为 k 均值算法提供了一个名为 tf.contrib.learn.KmeansClustering 的 API。k 均值算法将数据点分组在一起。借助这种算法，我们可以提取相册的嵌入并且一起找到个体的脸部，换句话说，可以聚集在一起。

6.3 小结

本章介绍了相似性学习的基础知识。我们研究了诸如度量学习、孪生网络和 FaceNet 等算法；学习了损失函数，如对比损失和三元组损失。本章还涵盖了两个不同的领域：排序和推荐。最后，通过了解检测、基准点检测和相似性评分等几个步骤，掌握了人脸识别的一系列过程。

下一章将介绍递归神经网络及其在自然语言处理问题中的应用。之后，我们将使用与图像模型相结合的语言模型为图像添加题注，讨论该问题的几种算法，并查看两种不同类型数据的实现。

第7章 图像题注

在本章中，我们将处理给图像加题注的问题。这涉及检测目标并为图像提供文本题注。图像题注也可以称为图像到文本的翻译。这曾经被认为是一个非常棘手的问题，现在已经有了相当好的解决方法。本章需要具有相应题注的图像数据集。我们将详细介绍图像题注的技术和应用。

本章涉及以下主题：

- 了解用于评估它们的不同数据集和指标；
- 了解用于自然语言处理问题的一些技术；
- 文本向量化模型；
- 几种图像题注算法；
- 不良的结果和改进的余地。

7.1 了解问题和数据集

为图像自动生成题注是一项关键的深度学习任务，因为它结合了语言和视觉这两个领域。该问题的独特性使其成为计算机视觉中的主要问题之一。图像题注的深度学习模型应该能够识别图像中存在的目标，并且以自然语言生成文本来表示目标与动作之间的关系。针对这个问题的数据集很少，其中最著名的是第4章中介绍的目标检测中COCO数据集的扩展。

7.2 理解图像题注的自然语言处理

由于必须从图像中生成自然语言，因此熟悉自然语言处理就显得非常重要了。自然语言处理的概念是一个很大的主题，因此我们将范围限制在与图像题注相关的主题上。文本是自然语言的一种形式。它是词或字符的序列。文本的原子元素称为标记（token），它是一个字符序列。一个字符是文本的一个原子元素。

为了以文本的形式处理自然语言，文本必须进行删除顿号、括号等预处理。然后，文本通过将它们分隔为空格而被标记为单词。接着，将这些单词转换为向量。接下来，我们将看到这个向量转换如何提供帮助。

7.2.1　用向量形式表达词

以向量形式表达的词可以帮助自身执行算术运算。向量必须是紧凑的，维度较小。同义词应该有相似的向量，反义词应该有不同的向量。可以将词转换为向量，以便比较关系，如图 7-1 所示。

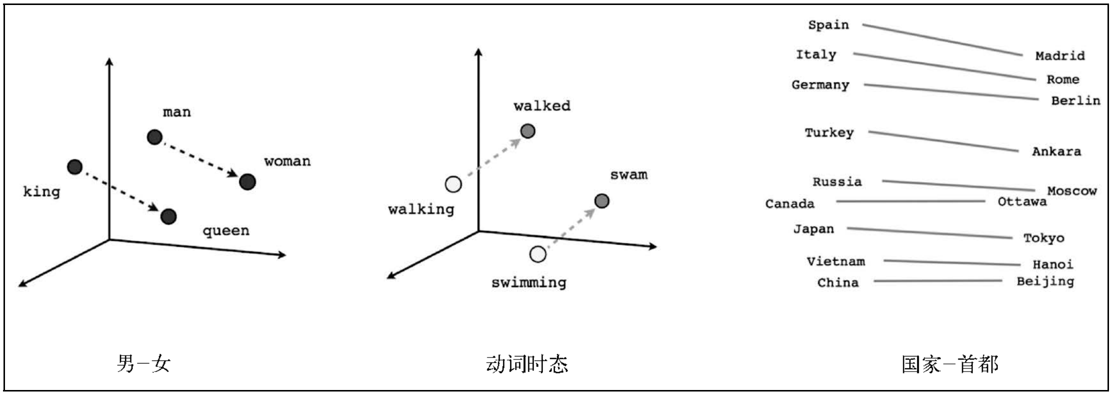


图　7-1

这种向量算法可以在不同实体之间的语义空间中进行比较。接下来，我们将学习创建一个可以将词转换为向量表示的深度学习模型。

7.2.2　将词转换为向量

通过在大型文本语料库上训练模型，可以将单词转换为向量。该模型经过训练，只要给出一个单词，就可以预测相近的单词。在预测相近单词的独热编码之前，词首先被独热编码，然后是隐藏层。以这种方式进行训练将创建一个紧凑的词表示。词的上下文可以通过两种方式获得，如下所示。

- Skip-gram：给定一个词，尝试预测几个接近的词。
- Continuous Bag Of Words（CBOW）：通过给定的一组词预测出一个词，是 Skip-gram 的反转。

图 7-2 说明了这些过程。

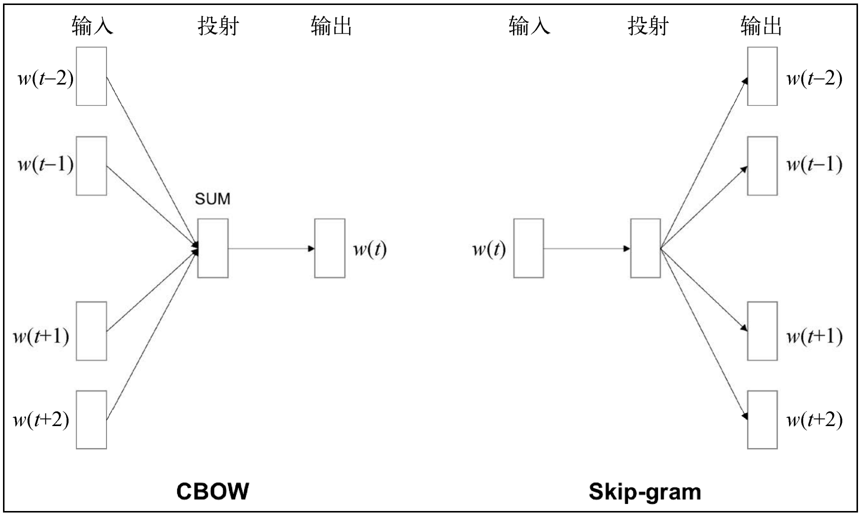


图　7-2

两种方法都显示出良好的效果。词被转换为嵌入空间中的向量。接下来介绍训练嵌入空间的细节。

7.2.3　训练一个嵌入

嵌入可以用图 7-3 所示的模型进行训练。

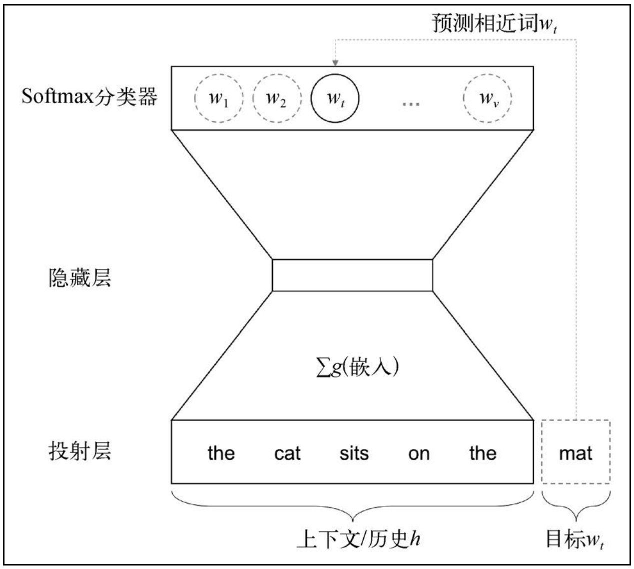


图　7-3

如图 7-3 所示，基于上下文或历史来预测目标词。预测基于 Softmax 分类器。隐藏层将嵌入学习为紧凑表示。请注意，这不是一个完整的深度学习模型，但它仍然运作良好。图 7-4 是使用 Softmax 分类器进行嵌入的低维可视化。

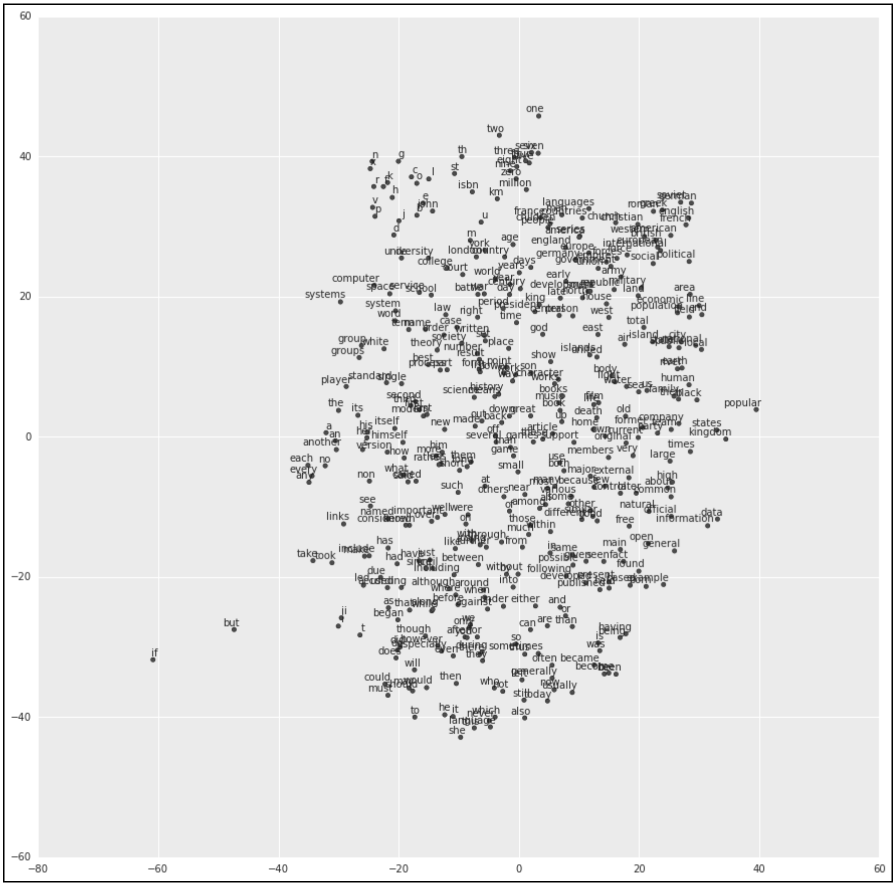


图 7-4

该可视化是使用 TensorBoard 生成的。具有相似语义但词性不同的词一起出现。

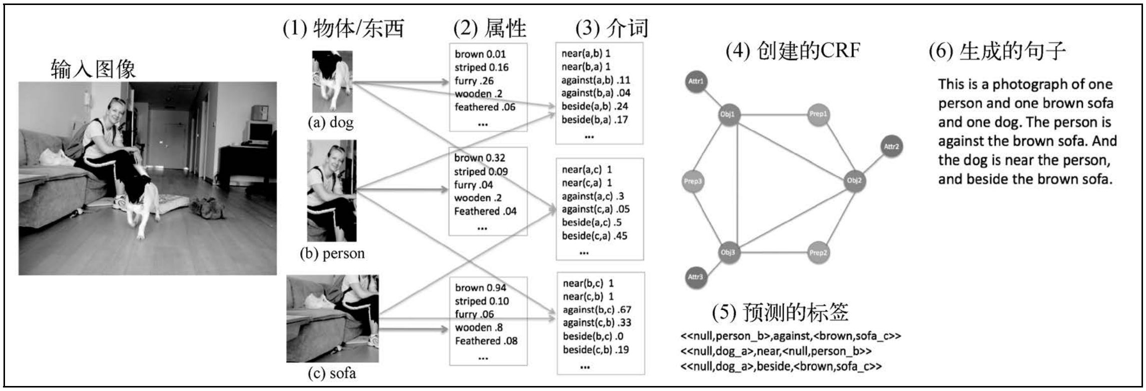
我们已经学会了如何训练用于生成文本的紧凑表示，接下来学习图像题注的方法。

7.3 图像题注和相关问题的方法

我们已经提出了几种用于题注图像的方法。直观地说，图像被转换为视觉特征并且从特征生成文本。生成的文本将以词嵌入的形式出现。生成文本的主要方法涉及 LSTM 和注意力（attention）。我们先从一种旧的文本生成方式开始。

7.3.1　使用条件随机场来链接图像和文本

Kulkarni 等人提出了一种从图像中查找目标和属性并用条件随机场（CRF）生成文本的方法。传统上，CRF 用于结构化预测，如文本生成等。生成文本的流程如图 7-5 所示。



* 经 Kulkarni 等人许可转载

图　7-5

在适当地放置介词以一致的方式生成文本方面，CRF 的使用存在局限。结果如图 7-6 所示。



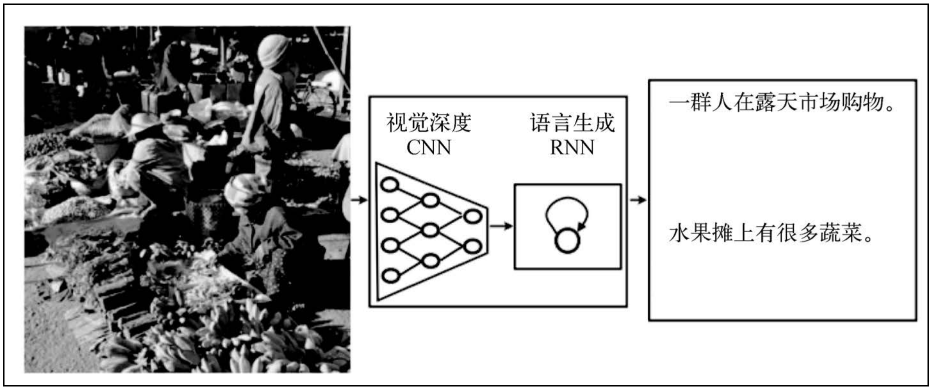
* 经 Kulkarni 等人许可转载

图　7-6

结果对目标和属性有适当的预测，但未能产生好的描述。

7.3.2　在 CNN 特征上使用 RNN 生成题注

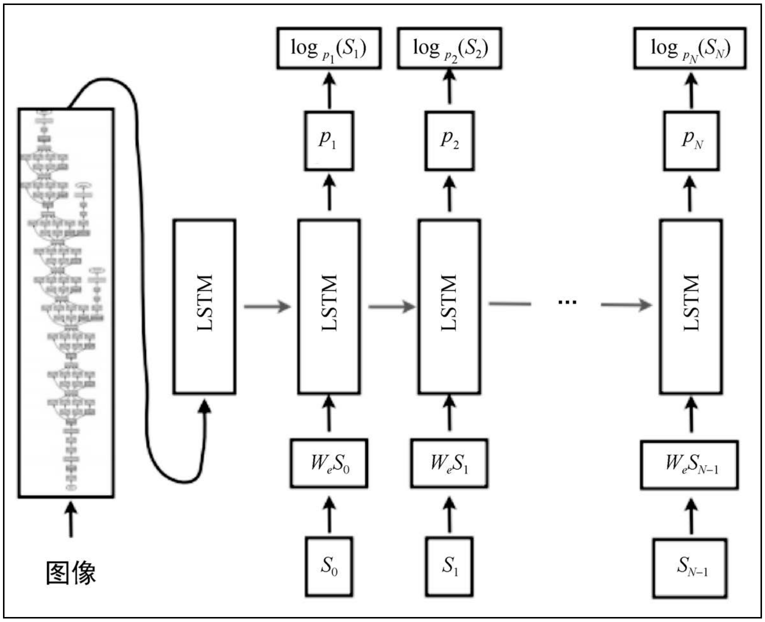
Vinyals 等人提出了一种端到端的图像题注的深度学习方法，该方法将 CNN 和 RNN 背靠背堆叠在一起。这是一个端到端的可训练模型，结构如图 7-7 所示。



* 经 Vinyals 等人许可转载（2015 年）

图　7-7

该模型可以生成一个以自然语言完成的句子。CNN 和 LSTM 的扩展视图如图 7-8 所示。



* 经 Vinyals 等人许可转载

图　7-8

图 7-8 是 LSTM 的扩展视图。图 7-9 显示了一组选择的结果。



* 经 Vinyals 等人许可转载（2015 年）

图　7-9

在这个过程中，CNN 将图像编码成特征，然后 RNN 根据这些特征生成一个句子。

7.3.3　使用图像排序创建题注

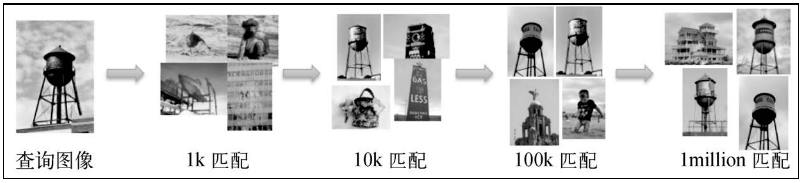
Ordonez 等人提出了一种方法，对图像进行排序，然后生成题注，流程如图 7-10 所示。



* 经 Ordonez 等人许可转载（2015 年）

图　7-10

从排序图像中提取的高级别信息可用于生成文本。可用于排序的图像越多，效果就越好，如图 7-11 所示。

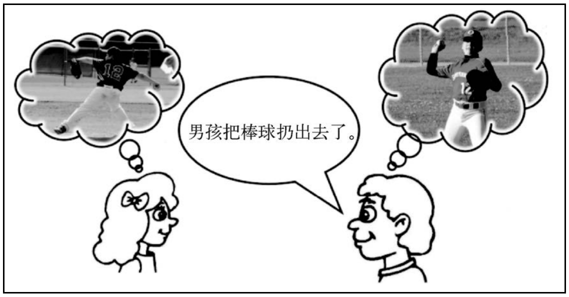


* 经 Ordonez 等人许可转载（2015 年）

图 7-11

7.3.4 从图像检索题注与从题注检索图像

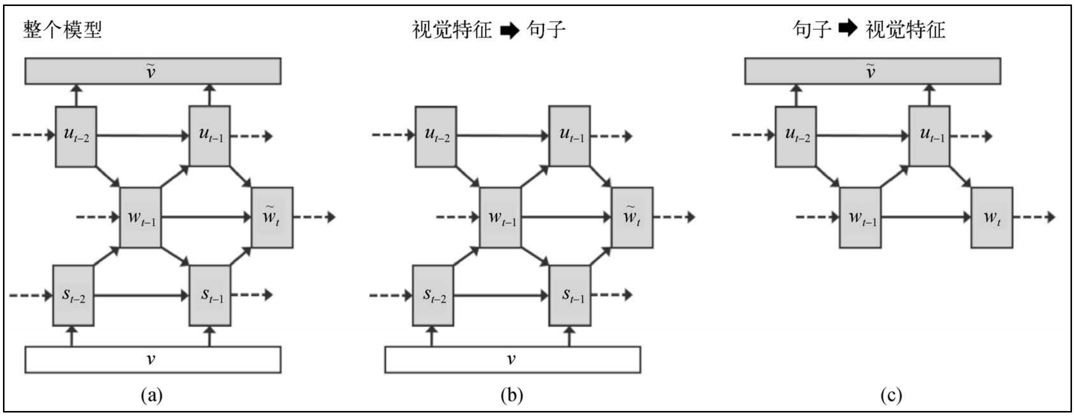
Chen 等人提出了一种从图像检索文本和从文本检索图像的方法。这是一个双向映射。图 7-12 展示了一个人用自然语言解释图像，而另一个人根据他的文本进行视觉想象。



* 经 Chen 等人许可转载（2015 年）

图 7-12

检索题注可以通过将图像和文字的编码器连接到一个隐空间（latent space）来实现，如图 7-13 所示。



* 经 Chen 等人许可转载（2015 年）

图 7-13

图像中的第一个模型是用于训练的完整模型。视觉特征也可用于生成句子，反之亦然。

7.3.5　密集题注

Johnson 等人提出了一种用于密集题注的方法。首先，让我们通过图 7-14 来了解这个任务。

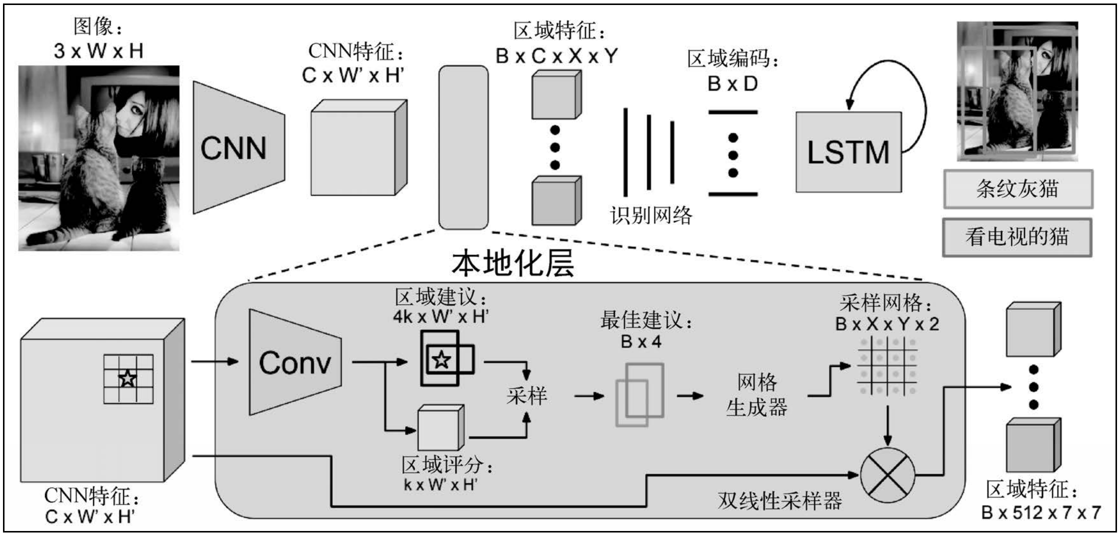


两个人站在海滩上。交通标志是黑色和白色。女孩抱着一个飞盘。一个木牌。白色标志与黑色字。
男子手持白色飞盘。空中的白色飞盘。短裤是蓝色的。支撑标志的金属杆。标志是黄色的。

图　7-14

如你所见，为图像中的目标和动作生成单独的标题，由此得名密集题注（dense captioning）。图 7-15 展示了 Johnson 等人提出的架构。

该架构实质上是 Faster-RCNN 和 LSTM 的组合。建立区域以生成目标检测结果，并使用区域的视觉特征来生成题注。

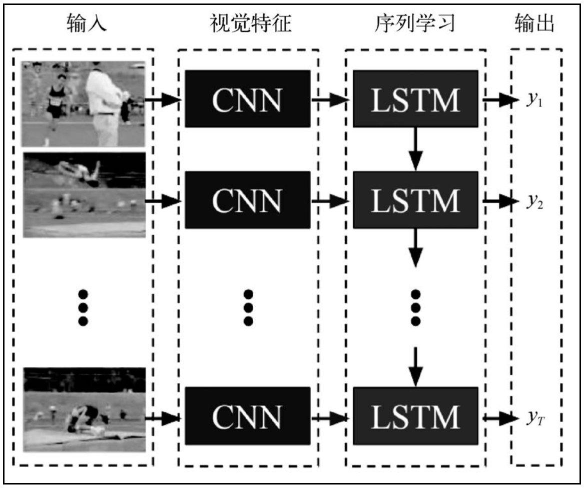


* 经 Johnson 等人许可转载

图 7-15

7.3.6 使用 RNN 生成题注

Donahue 等人提出了用于图像题注任务的**长期循环卷积架构**（long-term recurrent convolutional architecture，LRCN）。此模型的架构如图 7-16 所示。



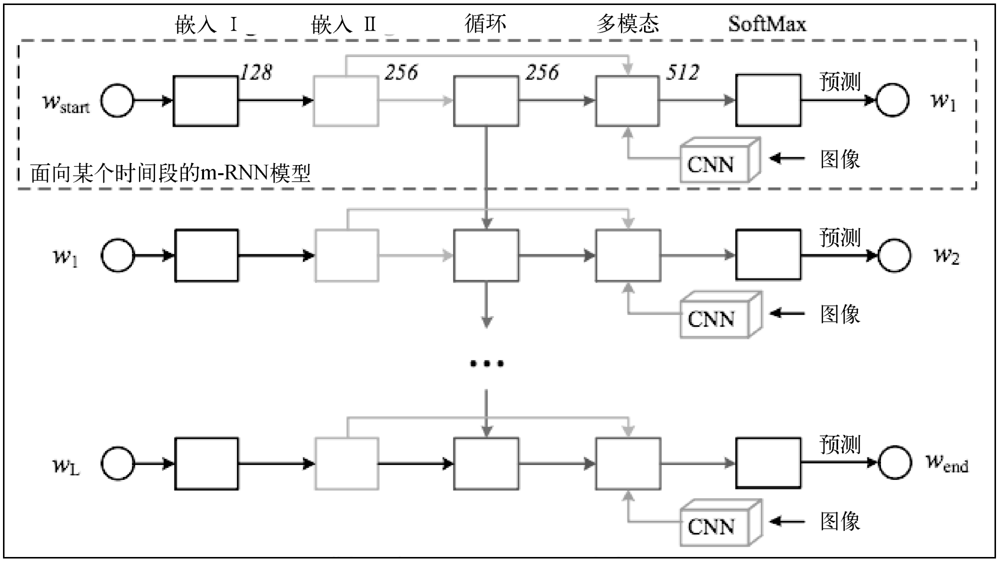
* 经 Donahue 等人许可转载

图 7-16

图中显示的 CNN 和 LSTM 都是在不同时间分享权重，这使得该方法可以扩展到任意长的序列。

7.3.7　使用多模态度量空间

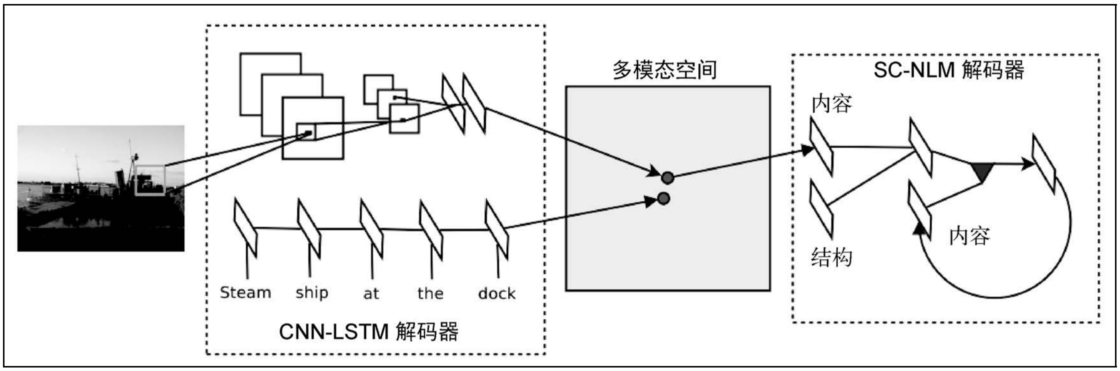
Mao 等人提出了**多模态嵌入空间**（multimodal embedding space）方法来生成题注，如图 7-17 所示。



* 经 Mao 等人许可转载

图　7-17

Kiros 等人提出了另一种多模态方法来生成题注，可以将图像和文本都嵌入到同一个多模态空间中，如图 7-18 所示。



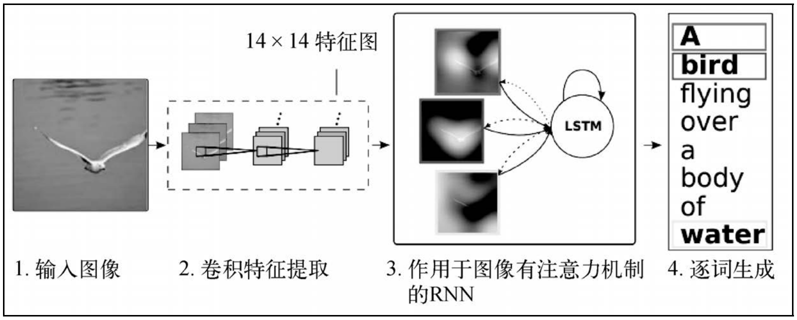
* 经 Kiros 等人许可转载

图　7-18

这两种多模态方法都有很好的效果。

7.3.8 使用注意网络生成题注

Xu 等人提出了一种使用注意力机制的图像题注的方法。 注意力机制赋予图像的某些区域更多的权重。注意力还可以实现可视化，向我们展示模型在生成下一个词时关注的位置。提出的模型如图 7-19 所示。



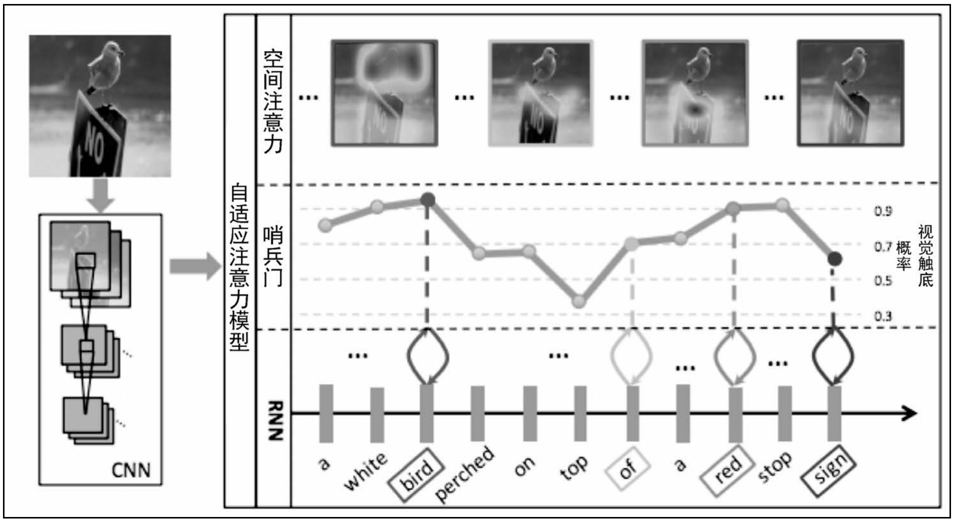
* 经 Xu 等人许可转载

图 7-19

首先，从图像中提取 CNN 特征。然后，将有注意力机制的 RNN 应用于图像而生成词。

7.3.9 知道什么时候查看

Lu 等人提出的注意力机制方法获得了优异的结果。知道什么时候查看注意力机制捕获的区域会产生更好的结果，流程如图 7-20 所示。

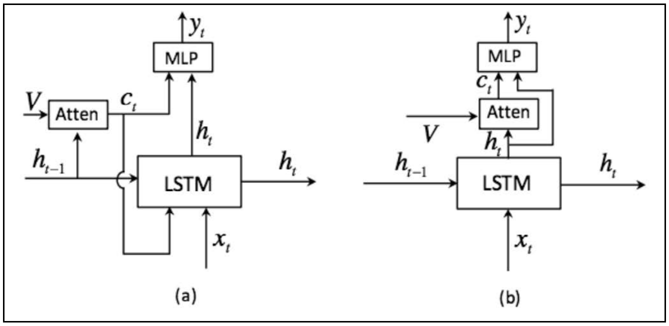


* 经 Lu 等人许可转载

图 7-20

7

注意力机制如图 7-21 所示。



* 经 Lu 等人许可转载

图 7-21

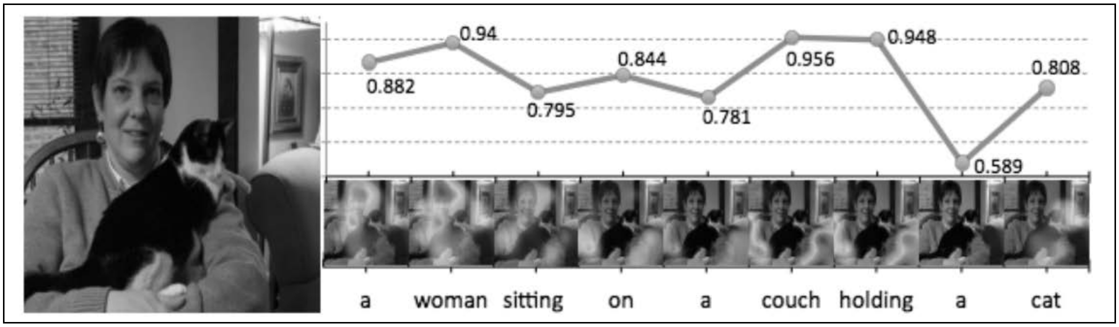
图 7-22 特别标示了注意力关注的区域的结果。



* 经 Lu 等人许可转载

图 7-22

在生成题注时，注意力的展开在这里可视化，如图 7-23 所示。



* 经 Lu 等人许可转载

图 7-23

我们已经学习了几种生成题注的方法，接下来来看一个实现。

7.4 实现基于注意力的图像题注

让我们使用以下代码从 VGG 和 LSTM 模型定义一个 CNN：

```
vgg_model = tf.keras.applications.vgg16.VGG16(weights='imagenet',
                                              include_top=False,
                                              input_tensor=input_tensor,
                                              input_shape=input_shape)
word_embedding = tf.keras.layers.Embedding(
    vocabulary_size, embedding_dimension, input_length=sequence_length)
embbedding = word_embedding(previous_words)
embbedding = tf.keras.layers.Activation('relu')(embbedding)
embbedding = tf.keras.layers.Dropout(dropout_prob)(embbedding)

cnn_features_flattened = tf.keras.layers.Reshape((height * height,
shape))(cnn_features)
net = tf.keras.layers.GlobalAveragePooling1D()(cnn_features_flattened)

net = tf.keras.layers.Dense(embedding_dimension, activation='relu')(net)
net = tf.keras.layers.Dropout(dropout_prob)(net)
net = tf.keras.layers.RepeatVector(sequence_length)(net)
net = tf.keras.layers.concatenate()([net, embbedding])
net = tf.keras.layers.Dropout(dropout_prob)(net)
```

现在已经定义了 CNN，接下来定义注意力层，如下所示：

```
h_out_linear = tf.keras.layers.Convolution1D(
    depth, 1, activation='tanh', border_mode='same')(h)
h_out_linear = tf.keras.layers.Dropout(
    dropout_prob)(h_out_linear)
h_out_embed = tf.keras.layers.Convolution1D(
    embedding_dimension, 1, border_mode='same')(h_out_linear)
z_h_embed = tf.keras.layers.TimeDistributed(
    tf.keras.layers.RepeatVector(num_vfeats))(h_out_embed)
```

```
Vi = tf.keras.layers.Convolution1D(
    depth, 1, border_mode='same', activation='relu')(V)

Vi = tf.keras.layers.Dropout(dropout_prob)(Vi)
Vi_emb = tf.keras.layers.Convolution1D(
    embedding_dimension, 1, border_mode='same', activation='relu')(Vi)

z_v_linear = tf.keras.layers.TimeDistributed(
    tf.keras.layers.RepeatVector(sequence_length))(Vi)
z_v_embed = tf.keras.layers.TimeDistributed(
    tf.keras.layers.RepeatVector(sequence_length))(Vi_emb)

z_v_linear = tf.keras.layers.Permute((2, 1, 3))(z_v_linear)
z_v_embed = tf.keras.layers.Permute((2, 1, 3))(z_v_embed)

fake_feat = tf.keras.layers.Convolution1D(
    depth, 1, activation='relu', border_mode='same')(s)
fake_feat = tf.keras.layers.Dropout(dropout_prob)(fake_feat)

fake_feat_embed = tf.keras.layers.Convolution1D(
    embedding_dimension, 1, border_mode='same')(fake_feat)
z_s_linear = tf.keras.layers.Reshape((sequence_length, 1,
depth))(fake_feat)
z_s_embed = tf.keras.layers.Reshape(
    (sequence_length, 1, embedding_dimension))(fake_feat_embed)

z_v_linear = tf.keras.layers.concatenate(axis=-2)([z_v_linear, z_s_linear])
z_v_embed = tf.keras.layers.concatenate(axis=-2)([z_v_embed, z_s_embed])

z = tf.keras.layers.Merge(mode='sum')([z_h_embed,z_v_embed])
z = tf.keras.layers.Dropout(dropout_prob)(z)
z = tf.keras.layers.TimeDistributed(
    tf.keras.layers.Activation('tanh'))(z)
attention = tf.keras.layers.TimeDistributed(
    tf.keras.layers.Convolution1D(1, 1, border_mode='same'))(z)
attention = tf.keras.layers.Reshape((sequence_length,
num_vfeats))(attention)
attention = tf.keras.layers.TimeDistributed(
    tf.keras.layers.Activation('softmax'))(attention)
attention = tf.keras.layers.TimeDistributed(
    tf.keras.layers.RepeatVector(depth))(attention)
attention = tf.keras.layers.Permute((1,3,2))(attention)
w_Vi = tf.keras.layers.Add()([attention,z_v_linear])
sumpool = tf.keras.layers.Lambda(lambda x: K.sum(x, axis=-2),
               output_shape=(depth,))
c_vec = tf.keras.layers.TimeDistributed(sumpool)(w_Vi)
atten_out = tf.keras.layers.Merge(mode='sum')([h_out_linear,c_vec])
h = tf.keras.layers.TimeDistributed(
tf.keras.layers.Dense(embedding_dimension, activation='tanh'))(atten_out)
h = tf.keras.layers.Dropout(dropout_prob)(h)

predictions = tf.keras.layers.TimeDistributed(
    tf.keras.layers.Dense(vocabulary_size, activation='softmax'))(h)
```

在前面代码的帮助下，我们定义了一个深度学习模型。该模型借助注意力机制将 CNN 特征与 RNN 结合起来。这是目前生成题注的最好方法。

7.5 小结

在本章中，我们理解了与图像题注相关的问题；学习了一些涉及自然语言处理的技术和各种 word2vec 模型，如 GLOVE；理解了几种算法，如 CNN2RNN、度量学习和组合目标；实现了一个结合了 CNN 和 LSTM 的模型。

下一章将介绍生成模型。我们将从头开始学习和实现风格算法，了解一些最佳模型，并学习很酷的**生成对抗网络**（GAN）及其各种应用。

第 8 章

生成模型

生成模型（generative model）已经成为计算机视觉中的重要应用。与之前章节中讨论的使用图像进行预测的应用不同，生成模型可以为特定目标创建图像。

本章涉及以下主题：

- 生成模型的应用；
- 风格迁移算法；
- 为超分辨率图像训练模型；
- 生成模型的实现和训练；
- 现有模型的缺点。

读完本章，你将能够实现一些很棒的迁移风格应用，并了解与生成模型相关的可能性以及困难。

8.1　生成模型的应用

我们首先讨论生成模型的可能应用。生成模型的应用非常多，我们将通过考查其中一些应用来理解其动机和可能性。

8.1.1　艺术风格迁移

艺术风格迁移是为任意图像进行艺术风格转化的过程。例如，可以将一幅图像的艺术风格与另一幅图像的内容相结合来创建图像。Gatys 等人展示了一个结合了几种不同风格的图像的例子（图 8-1）。图像 a 是原照片，应用不同风格的结果显示在其他图像中。



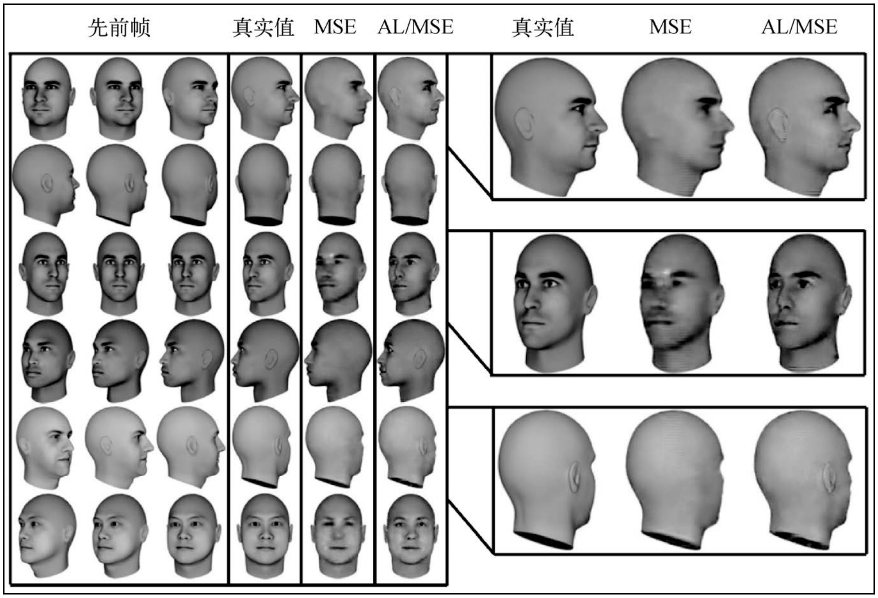
* 经 Gatys 等人许可转载

图 8-1

这个应用引起了公众的关注，市场上有几个移动应用程序提供这种功能。

8.1.2 预测视频中的下一帧

使用生成模型可以预测合成视频集的后续帧。图 8-2 由 Lotter 等人提出，其中左侧的图像是前一帧的模型，右侧是两种算法生成的图像与真实图像的比较。



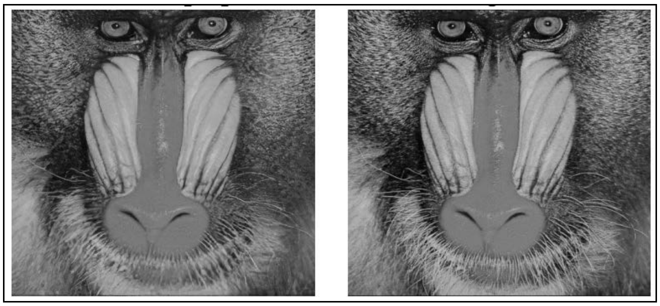
* 经 Lotter 等人许可转载

图　8-2

由生成模型生成的帧十分逼真。

8.1.3　图像的超分辨率

超分辨率是从较小的图像创建更高分辨率图像的过程。传统上，插值方法用于创建更大的图像。但是插值给出了平滑后的效果而忽略了高频细节。生成模型是为达到超分辨率这一特定目的而训练的，它可以创建具有优异细节的图像。图 8-3 是由 Ledig 等人提出的这种模型的一个例子。左侧以 4 倍的缩放比例生成，看起来与右侧的原始图像没有什么区别。



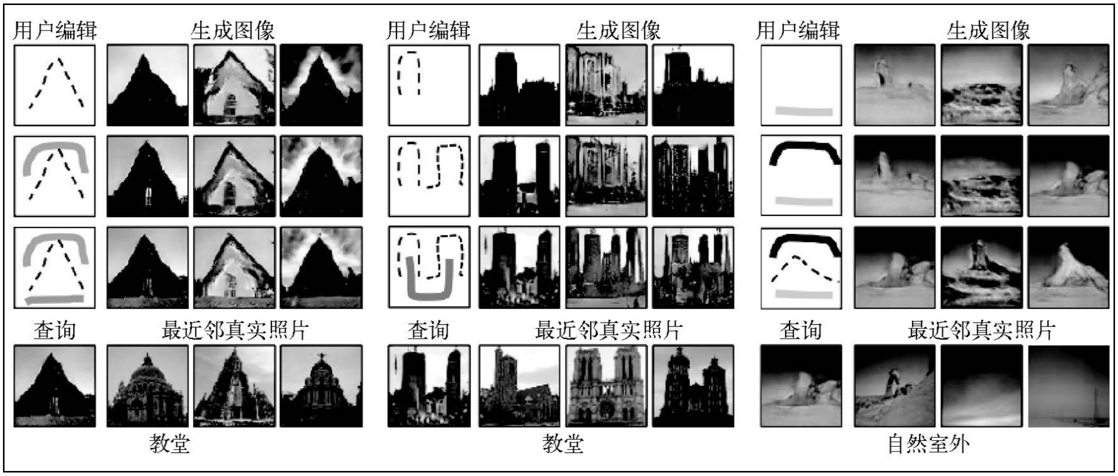
* 经 Ledig 等人许可转载

图　8-3

超高分辨率对于在高质量显示屏或打印机上显示低分辨率图像非常有用。另一种应用是压缩图像的高质量重建。

8.1.4 交互式图像生成

生成模型可以通过交互来创建图像。用户可以添加编辑（edit），然后生成图像。此处给出Zhu等人提出的编辑，如图 8-4所示。



* 经 Zhu 等人许可转载

图 8-4

如图 8-4 所示，图像是基于编辑的形状和颜色生成的。底部的一个绿色笔画创建为一片草地，一个矩形创建为一座摩天大楼等。通过用户的进一步输入生成这些图像并进行微调。生成的图像也可以用来检索可被使用的最相似的真实图像。交互式图像生成为直观地搜索图像提供了一种全新的方式。

8.1.5 图像到图像的翻译

图像可用于生成具有特定目标的其他图像，因此该过程称为**图像到图像的翻译**（image to image translation）。图 8-5 展示了翻译的几个例子，以及 Isola 等人提出的相应准则。



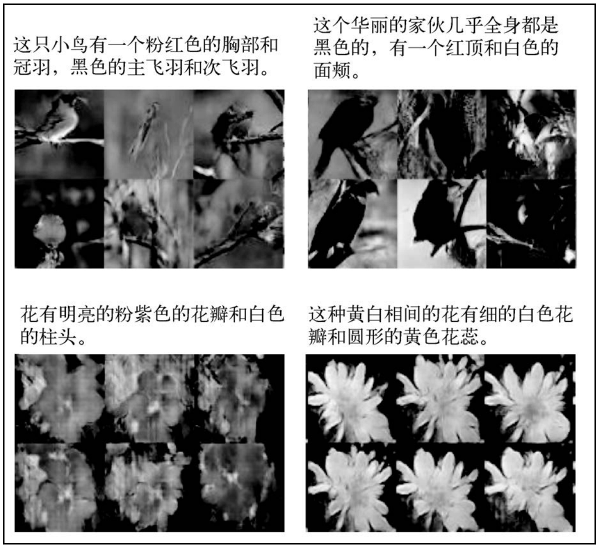
* 经 Isola 等人许可转载

图　8-5

带有标签的图形可以转换为具有创意的逼真的图像。黑白图像可以转换为彩色图像。这样的翻译对照片编辑应用程序、老电影着色、时尚设计等非常有用。

8.1.6　文本到图像的生成

图像可以从文本描述生成，步骤与图像到图像的翻译相似。图 8-6 是由 Reed 等人在自然文本描述中生成的一些例子。



* 经 Reed 等人许可转载

图　8-6

目前，该模型仅适用于少数几个目标。从文本生成图像还达不到在应用程序中使用的效果。

8.1.7 图像修复

修复（inpainting）是填补图像中的间隙的过程，如图 8-7 所示。

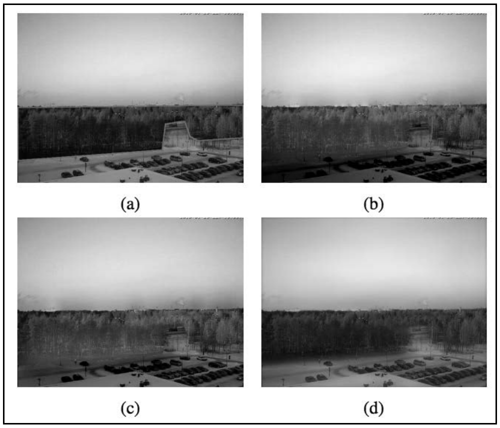


图 8-7

左侧是正常的图像，右侧是处理后的图像。正如图像中显示的那样，不需要的部分会从图像中删除。图像修复用于从图像中移除不需要的目标，以及填充扫描的艺术品的空间。

8.1.8 图像混合

图像混合（blending）是将图像的一部分没有任何人工痕迹地顺利粘贴到另一幅图像上的过程。图 8-8 中，图像 a 显示了一幅图像放在另一幅图像上的例子，效果却不好。图像 b 和图像 c 分别代表传统的混合技术，例如改进的泊松方法（modified Poisson method）和多样条方法（multi-spline method）。图像 d 显示了一种生成方法的混合结果。与其他方法相比，该方法的结果更好。



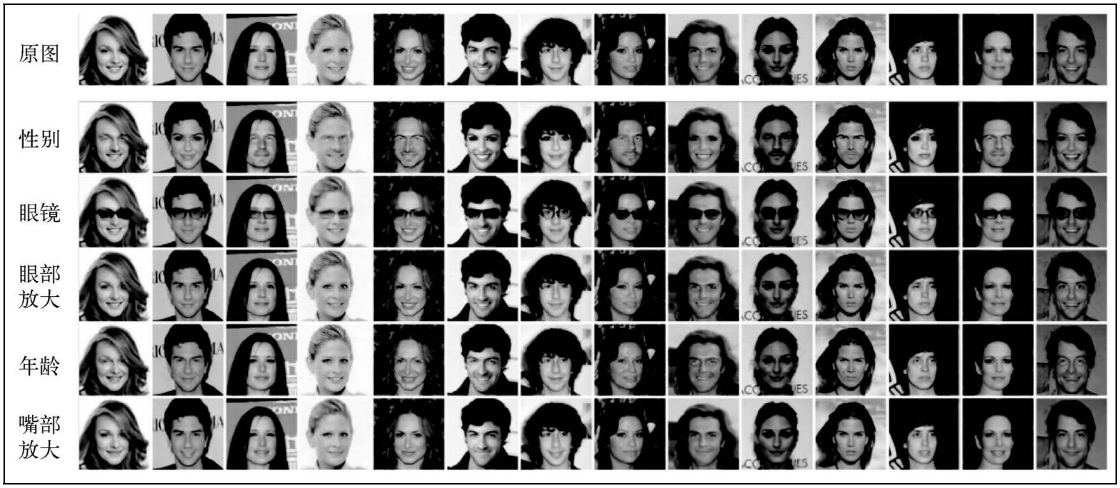
* 经 Wu 等人许可转载

图 8-8

图像混合在照片编辑和电影行业的特效中非常有用。

8.1.9　转换属性

图像的属性可以使用生成模型进行更改。人脸可以被修改以反映不同的属性，如性别、眼镜、年龄等，如图 8-9 所示（由 Lample 等人提出）。



* 经 Lample 等人许可转载

图　8-9

更改属性可面向创造性以及趣味性的应用，还可用于生成更多具有变化的训练数据。

8.1.10　创建训练数据

生成模型可以用于生成更大规模的训练集，甚至可以用于改善为训练创建的合成图像。图 8-10 是使用生成模型为交通标志识别而创建的合成图像（由 Wangle 等人提出）。



* 经 Wang 等人许可转载

图　8-10

使用这些图像可以使分类更加准确。

8.1.11 创建新的动画角色

如图 8-11 所示（由 Jin 等人提出），生成模型可用于创建具有各种属性的新的动画角色，如面部表情、发型、服饰等。



* 经 Jin 等人许可转载

图 8-11

创建具有不同属性的新角色可能会为动画行业带来革命性的改变。

8.1.12 照片 3D 模型

我们可以使用生成模型从 2D 图像创建 3D 模型，如图 8-12 所示（由 Wu 等人提出）。



* 经 Wu 等人许可转载

图 8-12

从图像创建 3D 模型对于机器人、增强现实和动画行业非常有用。我们将在下面的章节中学习它们背后的算法。在下一节中，我们将实现神经艺术风格迁移。

8.2 神经艺术风格迁移

要实现的第一个应用是**神经艺术风格迁移**（neural artistic style transfer）。在这里，我们把梵高的艺术风格迁移到一幅图像上。图像可以被视为风格和内容的组合。艺术风格转换技术将图像转换成看起来像具有特定绘画风格的绘画。我们将学习如何通过编码实现这个想法。`loss` 函数将比较用照片内容与绘画风格生成的图像。因此，要对图像像素进行优化，而不是对网络的权重进行优化。通过在照片的内容上与生成的图像进行比较，然后再根据绘画风格与生成的图像进行比较，计算出相应的两个值。

8.2.1 内容损失

由于像素不是一个好的选择，我们将使用各层的 CNN 特征，因为它们能更好地表示目标的内容。如第 3 章所述，初始层具有高频率特征，例如边缘、角落和纹理；但后面的层表示物体，因此对内容而言更好。后一层可以比像素更好地比较物体到物体。但为此，我们需要先使用以下代码导入所需的库：

```
import numpy as np
from PIL import Image
from scipy.optimize import fmin_l_bfgs_b
from scipy.misc import imsave
from vgg16_avg import VGG16_Avg
from keras import metrics
from keras.models import Model
from keras import backend as K
```

现在，使用以下命令加载所需的图像：

```
content_image = Image.open(work_dir + 'bird_orig.png')
```

我们将在这个实例中使用如图 8-13 所示的图像。



图 8-13

由于我们使用 VGG 架构来提取特征，因此必须从所有图像中减去所有 ImageNet 图像的均值，如下所示：

```
imagenet_mean = np.array([123.68, 116.779, 103.939], dtype=np.float32)


def subtract_imagenet_mean(image):
    return (image - imagenet_mean)[:, :, :, ::-1]
```

请注意，通道是不同的。preprocess 函数获取生成的图像并减去均值，然后反转通道。由于预处理步骤，deprocess 函数反转了该效果，如下所示：

```
def add_imagenet_mean(image, s):
    return np.clip(image.reshape(s)[:, :, :, ::-1] + imagenet_mean, 0, 255)
```

首先来看一下如何使用其他图像的内容创建一幅图像。这是一个从**随机噪声**中创建图像的过程。这里使用的内容是某些层中**激活值**的总和。我们将尽量减少随机噪声和图像之间内容的损失，这就是所谓的内容损失。这种损失类似于逐像素损失，但是应用在层的激活值上，因此将捕获到内容而排除噪声。任何 CNN 架构都可以用来对内容图像和随机噪声进行前向推断。激活值得到后计算均方误差，比较这两个输出的激活值。

随机图像的像素被更新而 CNN 权重被冻结。在该例子中，我们将冻结 VGG 网络。现在，可以加载 VGG 模型。生成图像对下采样技术（如**最大池化**）非常敏感。从最大池化中恢复像素值是不可能的。因此，**平均池化**比最大池化更平滑。使用平均池化转换 VGG 模型的函数用于加载模型，如下所示：

```
vgg_model = VGG16_Avg(include_top=False)
```

请注意，尽管池化类型已更改，但该模型的权重与原始模型的权重相同。ResNet 模型和 Inception 模型不适合这种情况，因为它们无法提供各种抽象。我们将采用 VGG 模型的最后一个卷积层即 block_conv1 的激活值，同时模型被冻结。这是 VGG 的倒数第三层，具有宽的感受野。这里给出代码供参考：

```
content_layer = vgg_model.get_layer('block5_conv1').output
```

现在，用截断的 VGG 创建一个新模型，直到提供良好特征的层。因此，图像现在可以加载并可用于执行前向推断，以得到实际激活的层。使用以下代码创建一个 TensorFlow 变量来捕获激活值：

```
content_model = Model(vgg_model.input, content_layer)
content_image_array =
subtract_imagenet_mean(np.expand_dims(np.array(content_image), 0))
content_image_shape = content_image_array.shape
target = K.variable(content_model.predict(content_image_array))
```

我们定义一个评估类来计算图像的损失和梯度。以下类会在迭代的任何点返回损失和梯度值：

```
class ConvexOptimiser(object):
    def __init__(self, cost_function, tensor_shape):
        self.cost_function = cost_function
        self.tensor_shape = tensor_shape
        self.gradient_values = None

    def loss(self, point):
        loss_value, self.gradient_values =
self.cost_function([point.reshape(self.tensor_shape)])
        return loss_value.astype(np.float64)

    def gradients(self, point):
        return self.gradient_values.flatten().astype(np.float64)
```

损失函数可以定义为特定卷积层激活值之间的均方误差。将在生成图像的层和原始内容照片之间计算损失，如下所示：

```
mse_loss = metrics.mean_squared_error(content_layer, target)
```

通过考虑模型的输入计算损失的梯度，如下所示：

```
grads = K.gradients(mse_loss, vgg_model.input)
```

函数的输入是模型的输入，函数的输出将是损失和梯度值的数组，如下所示：

```
cost_function = K.function([vgg_model.input], [mse_loss]+grads)
```

这个函数的优化是确定的，因此不需要 SGD。

```
optimiser = ConvexOptimiser(cost_function, content_image_shape)
```

此函数可以使用一个简单的优化器进行优化，因为它是凸函数且是确定的。我们也可以在迭代的每一步中保存图像。我们将以这样一种方式来定义它：最终优化的梯度值是可访问的，正如我们使用 scikit-learn 的优化器一样。请注意，这个损失函数是凸函数，因此，一个简单的优化器足够用于计算。优化器可以使用以下代码进行定义：

```
def optimise(optimiser, iterations, point, tensor_shape, file_name):
    for i in range(iterations):
        point, min_val, info = fmin_l_bfgs_b(optimiser.loss,
point.flatten(),
                                fprime=optimiser.gradients, maxfun=20)
        point = np.clip(point, -127, 127)
        print('Loss:', min_val)
        imsave(work_dir + 'gen_'+file_name+'_{i}.png',
add_imagenet_mean(point.copy(), tensor_shape)[0])
    return point
```

优化器获取 `loss` 函数、点和梯度这些参数，并返回更新。需要生成一幅随机图像，使内容损失最小化，如下所示：

```
def generate_rand_img(shape):
    return np.random.uniform(-2.5, 2.5, shape)/1
generated_image = generate_rand_img(content_image_shape)
```

图 8-14 是创建的随机图像。

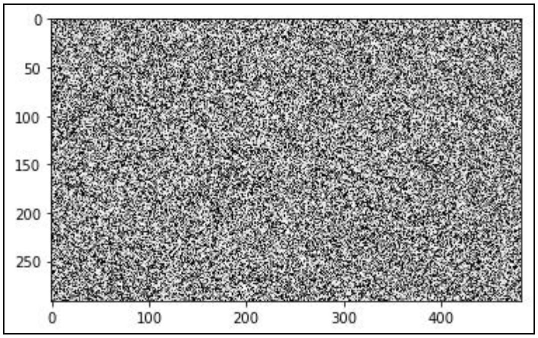


图 8-14

优化可以运行 10 次迭代以查看结果。

```
iterations = 10
generated_image = optimise(optimiser, iterations, generated_image,
content_image_shape, 'content')
```

如果一切顺利，在迭代过程中，损失应如下所示：

```
Current loss value: 73.2010421753
Current loss value: 22.7840042114
Current loss value: 12.6585302353
Current loss value: 8.53817081451
Current loss value: 6.64649534225
Current loss value: 5.56395864487
Current loss value: 4.83072710037
Current loss value: 4.32800722122
Current loss value: 3.94804215431
Current loss value: 3.66387653351
```

图 8-15 是生成的图像，现在它看起来差不多像一只鸟了。可以运行优化以进行进一步迭代来完成此操作。



图 8-15

优化器接收了图像并更新了像素，使内容相同。尽管结果较差，但它可以在一定程度上重现图像的内容。所有通过迭代得到的图像都直观地说明了如何生成图像。这个过程没有涉及批次。下一节中将介绍如何以绘画风格创建图像。

8.2.2　使用 Gram 矩阵的风格损失

创建具有原始图像内容的图像后，我们将了解如何仅使用风格创建图像。风格可以被认为是图像的颜色和纹理的混合。为此，我们将定义风格损失。首先，加载图像并将其转换为数组，如下所示：

```
style_image = Image.open(work_dir + 'starry_night.png')
style_image = style_image.resize(np.divide(style_image.size,
3.5).astype('int32'))
```

图 8-16 是我们加载的风格图像。

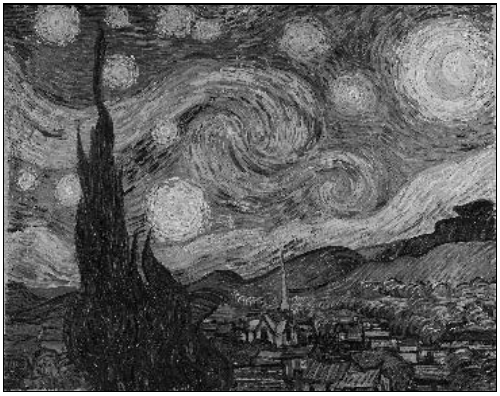


图　8-16

现在使用以下代码，通过更改通道来预处理此图像：

```
style_image_array = subtract_imagenet_mean(np.expand_dims(style_image,
0)[:, :, :, :3])
style_image_shape = style_image_array.shape
```

为此，我们将考虑几个层，就像在下面的代码中所做的那样：

```
model = VGG16_Avg(include_top=False, input_shape=shp[1:])
outputs = {l.name: l.output for l in model.layers}
```

接下来，使用以下代码将多个层作为前四个块的数组输出：

```
layers = [outputs['block{}_conv1'.format(o)] for o in range(1,3)]
```

创建一个新模型来输出所有这些层并分配目标变量，如下所示：

```
layers_model = Model(model.input, layers)
targs = [K.variable(o) for o in layers_model.predict(style_arr)]
```

风格损失使用 Gram 矩阵计算。Gram 矩阵是一个矩阵及其转置的乘积。激活值用来转置和相乘。然后用这个矩阵计算风格和随机图像之间的误差。Gram 矩阵丢失了位置信息，但会保留纹理信息。我们将使用下面的代码定义 Gram 矩阵：

```
def grammian_matrix(matrix):
    flattened_matrix = K.batch_flatten(K.permute_dimensions(matrix, (2, 0,
1)))
    matrix_transpose_dot = K.dot(flattened_matrix,
K.transpose(flattened_matrix))
    element_count = matrix.get_shape().num_elements()
    return matrix_transpose_dot / element_count
```

正如你知道的那样，它是衡量一对列之间的相关性的指标。高度和宽度维度变平坦。这不包括任何局部信息，因为坐标信息被忽略了。如以下代码所示，风格损失计算输入图像的 Gram 矩阵与目标之间的均方误差：

```
def style_mse_loss(x, y):
    return metrics.mse(grammian_matrix(x), grammian_matrix(y))
```

现在使用以下代码，将来自各个层的所有激活值相加来计算损失：

```
style_loss = sum(style_mse_loss(l1[0], l2[0]) for l1, l2 in
zip(style_features, style_targets))
grads = K.gradients(style_loss, vgg_model.input)
style_fn = K.function([vgg_model.input], [style_loss]+grads)
optimiser = ConvexOptimiser(style_fn, style_image_shape)
```

然后，创建一个随机图像来求解它，就像以前一样。但是这一次，我们还会应用一个高斯滤波器，如下所示：

```
generated_image = generate_rand_img(style_image_shape)
```

生成的随机图像如图 8-17 所示。

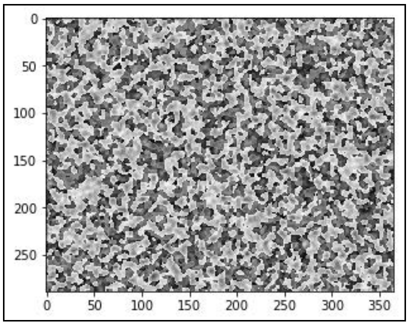


图 8-17

优化可以运行 10 次迭代以查看结果，如下所示：

```
generated_image = optimise(optimiser, iterations, generated_image,
style_image_shape)
```

如果一切顺利，求解器应打印类似以下内容的损失值：

```
Current loss value: 5462.45556641
Current loss value: 189.738555908
Current loss value: 82.4192581177
Current loss value: 55.6530838013
Current loss value: 37.215713501
Current loss value: 24.4533748627
Current loss value: 15.5914745331
Current loss value: 10.9425945282
Current loss value: 7.66888141632
Current loss value: 5.84042310715
```

图 8-18 是生成的图像。

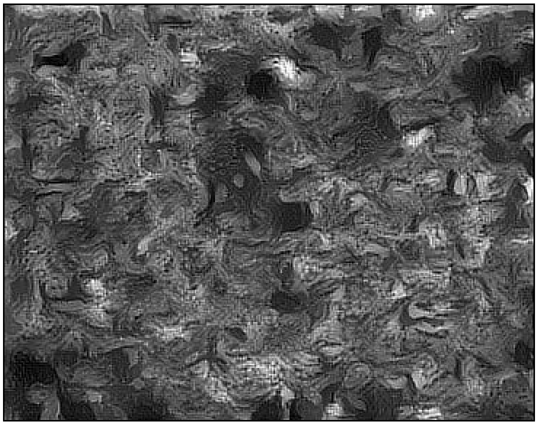


图　8-18

我们从随机噪声中创建了一幅具有特定绘画风格的图像，而没有任何位置信息。下一节将介绍如何将内容和风格损失相结合。

8.2.3　风格迁移

现在我们知道了如何重建图像，以及如何构建一幅能捕捉原始图像风格的图像。显而易见的想法是通过加权和添加两个 `loss` 函数将这两种方法结合起来，如下所示：

```
w,h = style.size
src = img_arr[:,:h,:w]
```

像以前一样，我们将获取一系列层的输出来计算风格损失。但是，我们仍然只需要一个层输

出来计算内容损失。如何知道要用哪个层？正如前面讨论的那样，层越低，内容重建就越精确。在将内容重建与风格融合在一起时，我们可能会期望对内容进行的宽松重建将为风格带来更多的空间（或灵感）。此外，后面的层可确保图像看起来像相同的主题，即使它没有相同的细节。此过程使用以下代码：

```
style_layers = [outputs['block{}_conv2'.format(o)] for o in range(1,6)]
content_name = 'block4_conv2'
content_layer = outputs[content_name]
```

现在，使用所需输出层创建独立风格模型，如下所示：

```
style_model = Model(model.input, style_layers)
style_targs = [K.variable(o) for o in style_model.predict(style_arr)]
```

使用内容层为内容创建另一个模型：

```
content_model = Model(model.input, content_layer)
content_targ = K.variable(content_model.predict(src))
```

现在，这两种方法的融合就像合并各自的损失函数一样简单。请注意，与之前的函数相反，此函数生成了三种不同类型的输出：

- 一幅用于原始图像；
- 一幅用于我们正在模拟的图像；
- 一幅用于我们正在训练其像素的随机图像。

调整重建混合的一种方式是改变内容损失的因子，我们在这里将其设为 1/10。如果增加分母，风格将对图像产生较大影响。如果它太大，图像的原始内容将被非结构化风格所遮蔽；同样，如果它太小，那么图像将没有足够的风格。在这个过程中使用如下代码：

```
style_wgts = [0.05,0.2,0.2,0.25,0.3]
```

`loss` 函数同时采用风格层和内容层，如下所示：

```
loss = sum(style_loss(l1[0], l2[0])*w
           for l1,l2,w in zip(style_layers, style_targs, style_wgts))
loss += metrics.mse(content_layer, content_targ)/10
grads = K.gradients(loss, model.input)
transfer_fn = K.function([model.input], [loss]+grads)
evaluator = Evaluator(transfer_fn, shp)
```

像以前一样运行求解器进行 10 次迭代，使用以下代码：

```
iterations=10
x = rand_img(shp)
x = solve_image(evaluator, iterations, x)
```

打印的损失值应如下所示：

```
Current loss value: 2557.953125
Current loss value: 732.533630371
Current loss value: 488.321166992
Current loss value: 385.827178955
Current loss value: 330.915924072
Current loss value: 293.238189697
Current loss value: 262.066864014
Current loss value: 239.34185791
Current loss value: 218.086700439
Current loss value: 203.045211792
```

这些结果是相当不错的。它们中的每一个都以艺术家的风格出色地重现了原始图像。生成的图像如图 8-19 所示。



图　8-19

现在结束风格迁移部分。此操作非常慢，但可以用在任何图像上。下一节将介绍如何使用类似的想法来创建超分辨率网络。以下几种方法可以使这一点变得更好：

- 向随机图像中添加一个高斯滤波器；
- 为一些层添加不同的权重；
- 将不同的层和权重用于内容；
- 初始化图像而不是初始化随机图像；
- 保持颜色；
- 将掩码（mask）用于所需的指定内容；
- 将草图转换成绘画；
- 绘制草图并创建图像。

TIP　任何图像都可以通过训练一个 CNN 转换为具有艺术风格的图像。

8.3 生成对抗网络

生成对抗网络（generative adversarial network，GAN）由 Ian Goodfellow 于 2014 年发明。它是一种无监督算法，两个神经网络分别作为鉴别器和生成器同时进行训练。该技术可以从随机噪声中生成图像，并由鉴别器评估是否为原始图像。经过进一步训练后，生成器网络可以生成照片般逼真的图像。生成器网络通常是一个反卷积神经网络，而鉴别器是一个卷积神经网络。

要理解这一点，有一个很好的比喻：将生成器想象为印刷假钞的人，而将鉴别器看作一个判断钱是否为假钞的警察。根据警察的反馈信息，生成器不断提高假钞的质量，直到警察无法区分真假。现在，让我们从实现开始。

8.3.1 vanilla GAN

最初的 GAN 被称为 vanilla GAN。在构建模型之前，定义一些对本章其余部分有用的层。以下是添加了泄露激活和正则化的 `convolution_layer`：

```
def convolution_layer(input_layer,
                      filters,
                      kernel_size=[4, 4],
                      activation=tf.nn.leaky_relu):
    layer = tf.layers.conv2d(
        inputs=input_layer,
        filters=filters,
        kernel_size=kernel_size,
        activation=activation,
        kernel_regularizer=tf.nn.l2_loss,
        bias_regularizer=tf.nn.l2_loss,
    )
    add_variable_summary(layer, 'convolution')
    return layer
```

然后，使用以下代码定义正则化的 `transpose_convolution_layer`，与 `convolution_layer` 相反：

```
def transpose_convolution_layer(input_layer,
                                filters,
                                kernel_size=[4, 4],
                                activation=tf.nn.relu,
                                strides=2):
    layer = tf.layers.conv2d_transpose(
        inputs=input_layer,
        filters=filters,
        kernel_size=kernel_size,
        activation=activation,
        strides=strides,
        kernel_regularizer=tf.nn.l2_loss,
        bias_regularizer=tf.nn.l2_loss,
    )
```

```
    add_variable_summary(layer, 'convolution')
    return layer
```

接下来，使用以下代码定义一个具有非线性激活的密集层：

```
def dense_layer(input_layer,
                units,
                activation=tf.nn.relu):
    layer = tf.layers.dense(
        inputs=input_layer,
        units=units,
        activation=activation
    )
    add_variable_summary(layer, 'dense')
    return layer
```

现在定义一个生成器，它将噪声作为输入并转换为图像。生成器由一对全连接层组成，然后是对噪声进行上采样的转置卷积层。最后，提出一个卷积层将噪声作为单个通道。每层之间都有批归一化层，可以使梯度平滑流动。我们将使用下面的代码来定义生成器：

```
def get_generator(input_noise, is_training=True):
    generator = dense_layer(input_noise, 1024)
    generator = tf.layers.batch_normalization(generator,
training=is_training)
    generator = dense_layer(generator, 7 * 7 * 256)
    generator = tf.layers.batch_normalization(generator,
training=is_training)
    generator = tf.reshape(generator,  [-1, 7, 7, 256])
    generator = transpose_convolution_layer(generator, 64)
    generator = tf.layers.batch_normalization(generator,
training=is_training)
    generator = transpose_convolution_layer(generator, 32)
    generator = tf.layers.batch_normalization(generator,
training=is_training)
    generator = convolution_layer(generator, 3)
    generator = convolution_layer(generator, 1, activation=tf.nn.tanh)
    return generator
```

现在定义 GAN 的鉴别器部分，该部分获得图像并试图区分伪造图像和真实图像。鉴别器是一个正常的卷积网络，其中有几个 `convolutional_layers`，后面是密集层。批归一化层存在于层之间。使用下面的代码来定义鉴别器：

```
def get_discriminator(image, is_training=True):
    x_input_reshape = tf.reshape(image, [-1, 28, 28, 1],
                                 name='input_reshape')
    discriminator = convolution_layer(x_input_reshape, 64)
    discriminator = convolution_layer(discriminator, 128)
    discriminator = tf.layers.flatten(discriminator)
    discriminator = dense_layer(discriminator, 1024)
    discriminator = tf.layers.batch_normalization(discriminator,
training=is_training)
    discriminator = dense_layer(discriminator, 2)
    return discriminator
```

鉴别器创建后，使用以下代码创建一个噪声向量，该向量将作为生成器的输入：

```
input_noise = tf.random_normal([batch_size, input_dimension])
```

GAN 模型可以使用 TensorFlow 中的 `tf.contrib.gan` 模块创建。它采用生成器和鉴别器方法及其相应的输入，如下所示：

```
gan = tf.contrib.gan.gan_model(
    get_generator,
    get_discriminator,
    real_images,
    input_noise)
```

现在，可以使用 `gan_train` 方法开始训练。该方法用于生成器和鉴别器的训练，使用了带有损失和优化器的 `gan_train_ops` 方法。

```
tf.contrib.gan.gan_train(
    tf.contrib.gan.gan_train_ops(
        gan,
        tf.contrib.gan.gan_loss(gan),
        tf.train.AdamOptimizer(0.001),
        tf.train.AdamOptimizer(0.0001)))
```

通过运行这个方法，一个 GAN 模型创建出来了。它可以从随机向量输出图像。生成的图像不受限制，可以来自任何标签。在下一节中，我们将使用一个条件 GAN 来产生想要的输出。

8.3.2 条件 GAN

条件 GAN（conditional GAN）可以生成带有我们想要的标签的图像。例如，我们想要让模型生成数字 8，模型就会生成一个 8。为了实现这一点，需要标签和噪声一起进行模型训练，如下所示：

```
gan = tf.contrib.gan.gan_model(
    get_generator,
    get_discriminator,
    real_images,
    (input_noise, labels))
```

其余的训练与 vanilla GAN 相似。接下来，我们将使用 GAN 来压缩图像。

8.3.3 对抗损失

对抗损失（adversarial loss）是生成器的损失。这种损失可以与虚假图像和真实图像之间的像素级的损失相结合，形成组合的对抗损失。`real_images` 需要同时提供给 GAN 模型的生成器和鉴别器，如下所示：

```
gan = tf.contrib.gan.gan_model(
    get_autoencoder,
    get_discriminator,
```

```
    real_images,
    real_images)
```

生成器是一个自编码器。其实现可以在第 3 章中找到。在此之后，我们将使用以下代码来定义损失：

```
loss = tf.contrib.gan.gan_loss(
    gan, gradient_penalty=1.0)

l1_pixel_loss = tf.norm(gan.real_data - gan.generated_data, ord=1)

loss = tf.contrib.gan.losses.combine_adversarial_loss(
    loss, gan, l1_pixel_loss, weight_factor=1)
```

GAN 损失的梯度受到惩罚。然后，计算像素级的损失并将其添加到惩罚性损失中。训练此模型创建一个强大的自编码器，可用于图像压缩。

8.3.4　图像翻译

一幅图像可以被翻译成另一幅图像，正如我们在应用部分已经学到的那样。创建 GAN 模型时，输入图像提供给鉴别器，而目标图像提供给生成器，如下所示：

```
gan = tf.contrib.gan.gan_model(
    get_generator,
    get_discriminator,
    real_images,
    input_images)
```

除了使用像素级的损失训练模型，还可以使用最小平方损失。可以使用以下代码进行计算：

```
loss = tf.contrib.gan.gan_loss(
    gan,
    tf.contrib.gan.losses.least_squares_generator_loss,
    tf.contrib.gan.losses.least_squares_discriminator_loss)

l1_loss = tf.norm(gan.real_data - gan.generated_data, ord=1)

gan_loss = tf.contrib.gan.losses.combine_adversarial_loss(
    loss, gan, l1_loss, weight_factor=1)
```

使用这种技术，一幅图像可以被翻译成另一幅图像。

8.3.5　InfoGAN

InfoGAN 可以生成所需标签的图像，而无须任何显式的监督训练。`infogan_model` 采用非结构化和结构化的输入，如下所示：

```
info_gan = tf.contrib.gan.infogan_model(
    get_generator,
    get_discriminator,
```

```
    real_images,
    unstructured_input,
    structured_input)

loss = tf.contrib.gan.gan_loss(
    info_gan,
    gradient_penalty_weight=1,
    gradient_penalty_epsilon=1e-10,
    mutual_information_penalty_weight=1)
```

由于训练不稳定，损失的定义带有一个惩罚项。在训练中增加惩罚可以提供更多的稳定性。

8.3.6 GAN 的缺点

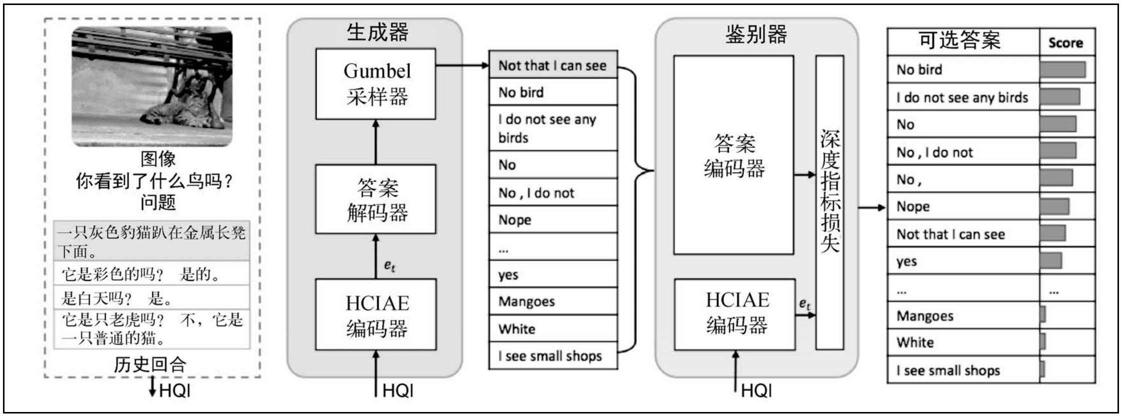
GAN 生成的图像在一些方面有缺点，如计数、视角和全局结构。目前正在进行广泛的研究，以改进模型。

8.4 视觉对话模型

视觉对话模型（visual dialogue model，VDM）支持基于图像的聊天。VDM 应用了计算机视觉、自然语言处理和聊天机器人技术。它已经找到了主要的应用，例如向盲人解释图像、向医生解释医学扫描、虚拟伴侣，等等。接下来，我们将学习解决这个问题的算法。

VDM 算法

Lu 等人提出了基于 GAN 的 VDM 算法。生成器生成答案，鉴别器对这些答案进行排序。图 8-20 是该过程的示意图。



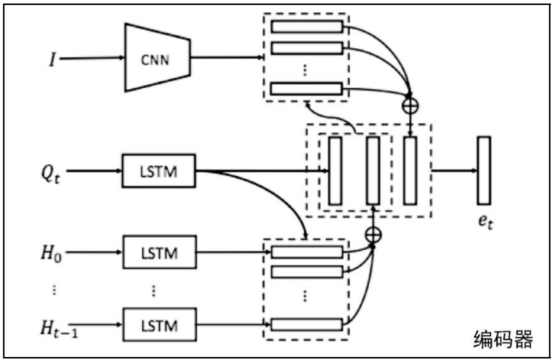
* 经 Lu 等人许可转载

图 8-20

聊天历史、当前问题和图像作为输入传送给生成器。接下来介绍生成器的工作原理。

1. 生成器

生成器有一个编码器和解码器。编码器将图像、问题和历史记录作为输入。编码器首先参照 LSTM 的历史记录，并参照图像的输出。流程如图 8-21 所示。



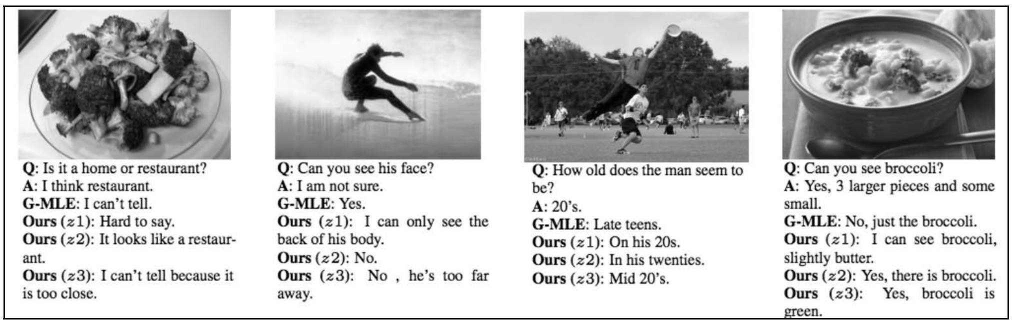
* 经 Lu 等人许可转载

图 8-21

整个历史记录都是可用的，并且 LSTM 参照聊天的历史记录。输出由产生嵌入的图像处理。由编码器生成的嵌入被解码器用来创建答案。解码器由 RNN 组成。编码器和解码器一起组成生成器，产生可能的答案。接下来介绍鉴别器是如何工作的。

2. 鉴别器

鉴别器从生成器获取生成的序列并对它们进行排序。排名是通过 n 对损失学习的嵌入来完成的。除了使用 n 对正负值进行比较之外，n 对损失类似于三元组损失。图 8-22 是该模型产生的一些结果。



* 经 Lu 等人许可转载

图 8-22

这样做的结果比一个简单的鉴别器产生的结果合理并且更好。

8.5 小结

在本章中，我们学习了生成模型和大量的应用。我们实现了它们，在保留内容的同时从一种图像风格转移到另一种，理解了 GAN 背后的直觉并训练模型达到相同的效果。最后，我们了解了视觉对话系统。

下一章介绍视频分析的深度学习方法。我们将了解如何通过相机、文件等访问视频内容，然后通过在视频帧级和视频整体上应用分类来实现视频分类。我们还会了解如何跟踪视频中的目标。

第 9 章

视频分类

在本章中，我们将学习如何为视频数据训练深度学习模型。我们将在帧的基础上开始对视频进行分类，然后使用时间信息来获得更好的准确率，之后将图像的应用扩展到视频，包括姿态估计、题注和生成视频。

本章涉及以下主题：

- 视频分类的数据集和算法；
- 将视频分成多个帧并对视频进行分类；
- 在单个帧级别 0 上为视觉特征训练模型；
- 了解 3D 卷积及其在视频中的使用；
- 在视频中结合运动向量；
- 利用时间信息进行目标跟踪；
- 人体姿态估计和视频题注等应用。

9.1 了解视频和视频分类

视频不过是一系列的图像。视频在时间方向上为图像带来了新的维度。将图像的空间特征和视频的时间特征放在一起，可以得到比仅用图像更好的结果。额外的维度也产生了很大的空间，因此增加了训练和推断的复杂性。处理视频的计算需求非常高。视频还改变了深度学习模型的架构，因为我们必须考虑时间特征。

视频分类任务是给视频标记类别。类别可以在帧级别上，也可以在整个视频上。视频中可能会执行某些动作或任务。因此，视频分类可以标记视频中出现的目标或者视频中发生的动作。下一节将介绍可用于视频分类任务的数据集。

9.1.1 探索视频分类数据集

视频分类是视频数据研究的主要问题。拍摄一些视频并标记与数据相关的各种目标或动作。

数据集根据标签的规模、质量和类型而有所不同。有些数据集甚至包含多个视频标签。这些视频通常很短。长视频可能会执行许多动作，因此在对视频片段进行分类之前会先在时间上把视频分割开。接下来，考虑一些特定数据集的细节。

1. UCF101

UCF101 是一个动作识别数据集。这些视频是在 YouTube 上收集的，由实际的动作组成。该数据集中有 101 个动作类别。还有一个名为 UCF50 的数据集，它有 50 个类别。UCF101 数据集中有 13 320 个视频。视频具有背景、缩放、姿态、遮挡和照明条件的多样变化。动作类别分为 25 个，也分别具有相似的变化，诸如背景、姿态、缩放、视点和照明等。

图 9-1 显示了动作名称及其相应的视频数量。

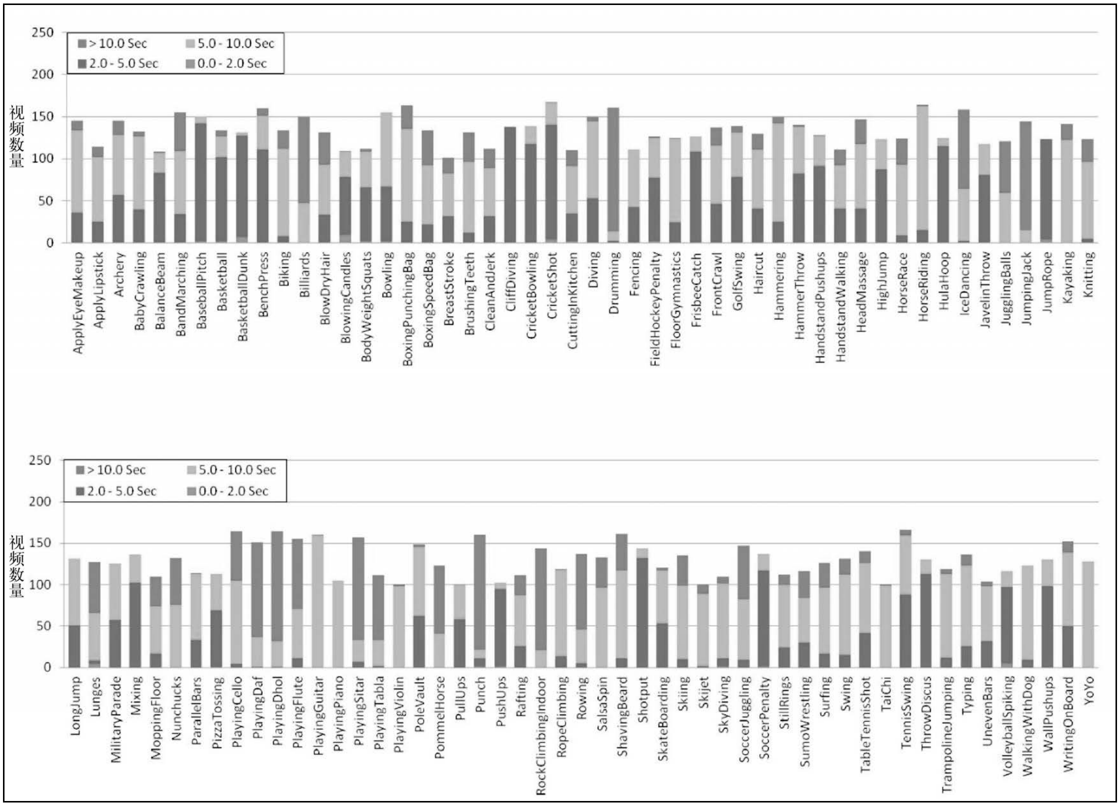


图　9-1

所有 101 个动作可以分为五种类型：人与物交互，肢体动作，人与人交互，演奏乐器和运动。

接下来，我们将介绍 YouTube-8M 数据集。

2. YouTube-8M

YouTube-8M 数据集适用于视频分类问题。该数据集包含带有标签和视觉特征的视频网址。以下是关于 YouTube-8M 数据集的一些统计数据。

- 视频网址数量：700 万。
- 视频片段时长（小时）：450 000。
- 类别标签数量：4716。
- 每个视频的平均标签数量：3.4。

图 9-2 是各种类型数据集的总结。

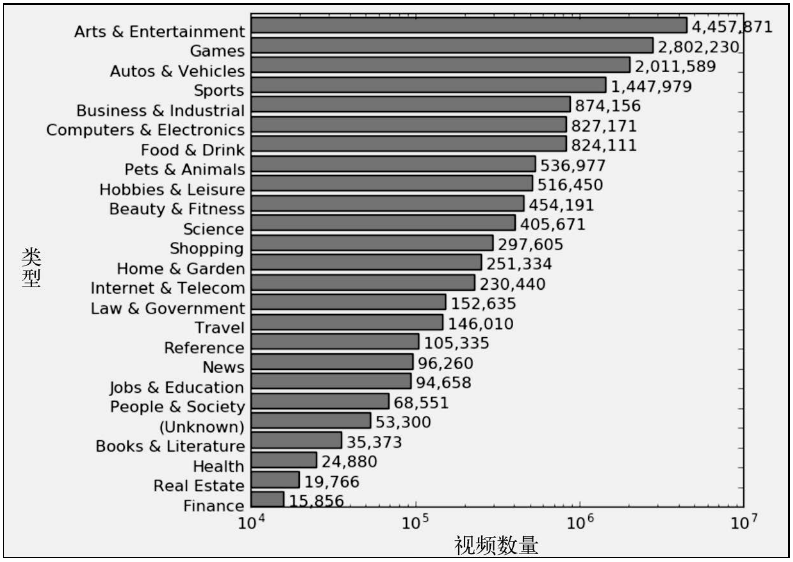


图　9-2

图 9-2 显示了数据集中可用标签的类型。视频数据很大，因此计算的视觉特征与数据集一起提供。

3. 其他数据集

还有一些数据集可用于视频分类问题。以下是几个数据集的详细信息。

- Sports-1M（Sports -1 Million）：拥有 1 133 158 个视频，487 个类别。标注是自动产生的。
- UCF-11（University of Central Florida -11 actions）：拥有 1600 个视频和 11 个动作。这些视频每秒帧数为 29.97。数据集可以与 UCF101 一起下载。
- HMDB-51（Human Motion DataBase -51 actions）：拥有 5100 个视频，51 个动作。
- Hollywood2：拥有 1707 个视频，12 个动作。

我们已经了解了可用于视频分类任务的数据集，以及描述和访问链接。接下来介绍如何加载视频并将其分割为帧以进一步处理。

9.1.2　将视频分割成帧

视频可以转换为很多帧，保存在目录中供进一步使用。分割成帧可帮助我们在训练过程之前解压缩视频，从而节省时间。首先，我们来看看用于将视频转换为帧的代码片段。

```
import cv2
video_handle = cv2.VideoCapture(video_path)
frame_no = 0
while True:
  eof, frame = video_handle.read()
  if not eof:
      break
  cv2.imwrite("frame%d.jpg" % frame_no, frame)
  frame_no += 1
```

使用这个片段，所有前面的数据集都可以转换为帧。请注意，这将需要大量的硬盘空间。

9.1.3　视频分类方法

视频必须为一些应用分类。由于视频中包含大量数据，因此必须考虑训练和推断的计算量。所有的视频分类方法都受到图像分类算法的启发。VGG、Inception 等标准架构可以用于帧级别的特征计算，然后进一步处理。在前面的章节中学到的 CNN、注意力机制和 LSTM 等概念在这里将会很有用。直观地，以下方法可用于视频分类。

- 提取帧并使用第 2 章中学到的模型进行帧分类。
- 提取在第 3 章中学习过的图像特征，并且这些特征可以用于训练 RNN，如第 7 章中所述。
- 在整个视频上训练一个 3D 卷积网络。3D 卷积是 2D 卷积的扩展；在下面的几节中将详细介绍 3D 卷积的工作原理。
- 使用视频的光流（optical flow）来进一步提高准确性。光流是目标运动的模式，在接下来的几节中将进行详细介绍。

下面几种算法可以提供很好的准确度，并具有各种计算复杂度。可以通过将视频转换为帧并将其下采样到相同的长度来准备数据集。某些预处理可以提供帮助，例如减去 Imagenet 的平均值。

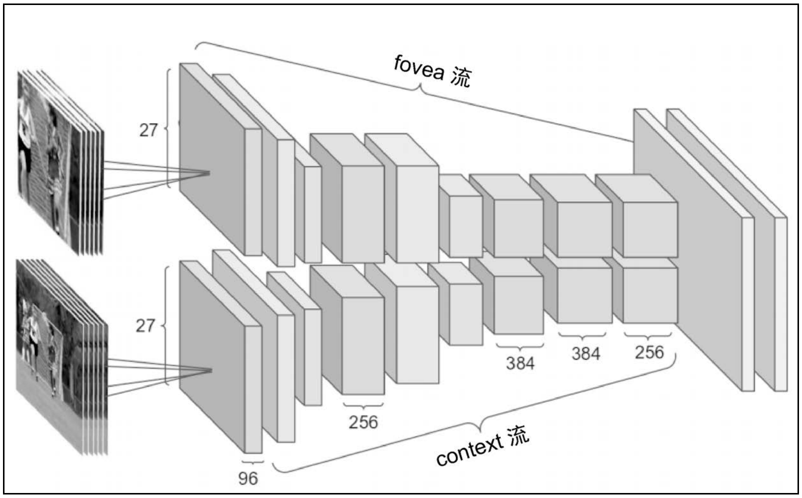
1. 为视频分类融合并行 CNN

从帧级别来看，由于图像的下采样会损失细节，视频的预测可能不会产生良好的结果。使用高分辨率 CNN 会增加推断时间。因此，Karpathy 等人提出将两个并行运行的视频流进行视频分

类。做帧预测有两个问题，即：

- 由于更大的 CNN 架构，预测可能需要很长时间；
- 独立预测会丢失沿时间维度的信息。

使用两个较小的编码器并行运行，可以用更少的参数简化架构。视频同时通过两个 CNN 编码器传递。一个编码器接收低分辨率并处理高分辨率。编码器具有交替的卷积、归一化和池化层。两个编码器的最后一层通过全连接层连接。另一个编码器大小相同，但只使用中心裁剪，如图 9-3 所示。



* 经 Karpathy 等人许可转载

图　9-3

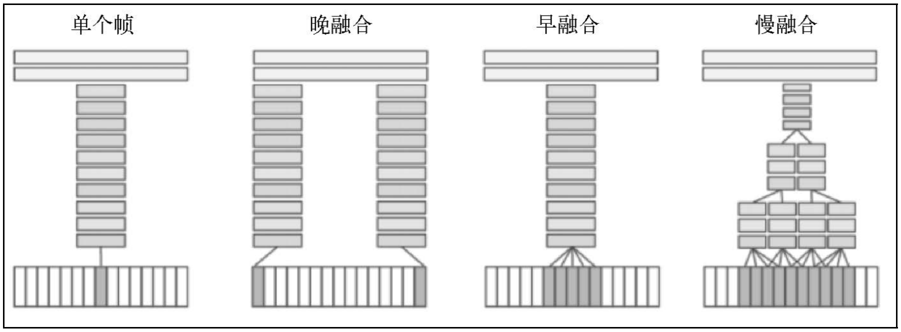
通过对视频进行下采样，帧的并行处理使运行更快。CNN 架构在保持相同精确度的同时将参数减半。这两个流被称为 fovea 流和 context 流。这些流显示在以下代码片段中：

```
high_resolution_input = tf.placeholder(tf.float32, shape=[None,
input_size])
low_resolution_input = tf.placeholder(tf.float32, shape=[None, input_size])
y_input = tf.placeholder(tf.float32, shape=[None, no_classes])
high_resolution_cnn = get_model(high_resolution_input)
low_resolution_cnn = get_model(low_resolution_input)
dense_layer_1 = tf.concat([high_resolution_cnn, low_resolution_cnn], 1)
dense_layer_bottleneck = dense_layer(dense_layer_1, 1024)
logits = dense_layer(dense_layer_bottleneck, no_classes)
```

沿时间维度处理的帧如图 9-4 所示。

视频可以在不同的时间看到，而不是只能看到固定大小的几个视频片段。在前面的图像中介绍了连接时间信息的三种方法。晚融合需要更长的时间，而早融合只需要几帧。慢融合结合了晚融合和早融合的优点，取得了良好的效果。该模型在 Sports1M 数据集上进行训练，该数据集有

487个类别，并且达到了50%的准确性。同样的模型应用于UCF101时，可以达到60%的准确率。



* 经 Karpathy 等人许可转载

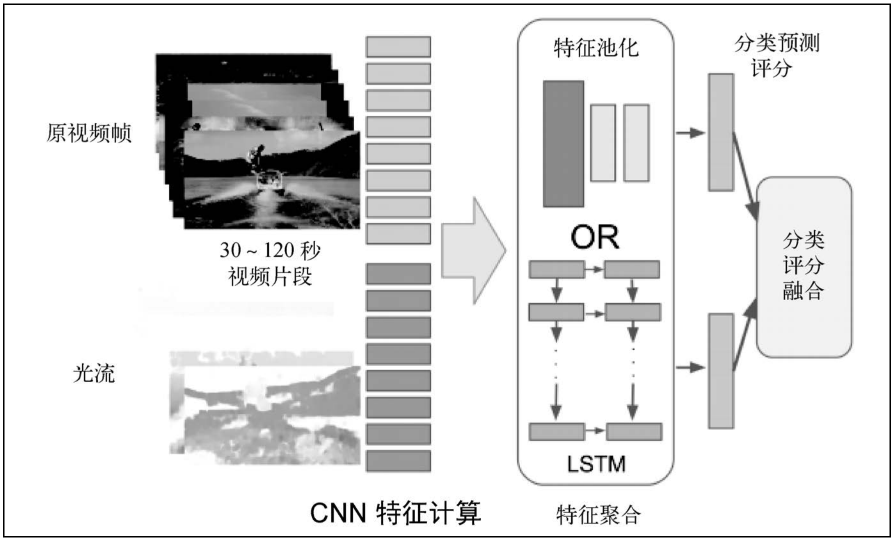
图 9-4

2. 长时间分类视频

融合方法适用于短视频片段，对较长的视频进行分类很困难，因为必须计算和记住很多帧。Ng等人提出了两种分类较长视频的方法。

- 第一种方法是在时间上池化卷积特征。最大池化作为一个特征聚合方法。
- 第二种方法是用一个LSTM连接卷积特征来处理视频的可变长度。

图9-5显示了这两种方法。



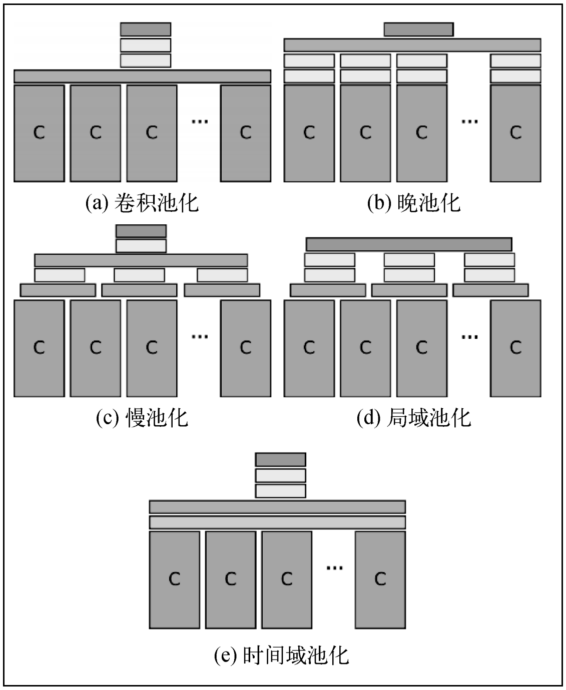
* 经 Ng 等人许可转载

图 9-5

CNN 特征可以被提取并送到一个小的 LSTM 网络，如以下代码所示：

```
net = tf.keras.models.Sequential()
net.add(tf.keras.layers.LSTM(2048,
            return_sequences=False,
            input_shape=input_shape,
            dropout=0.5))
net.add(tf.keras.layers.Dense(512, activation='relu'))
net.add(tf.keras.layers.Dropout(0.5))
net.add(tf.keras.layers.Dense(no_classes, activation='softmax'))
```

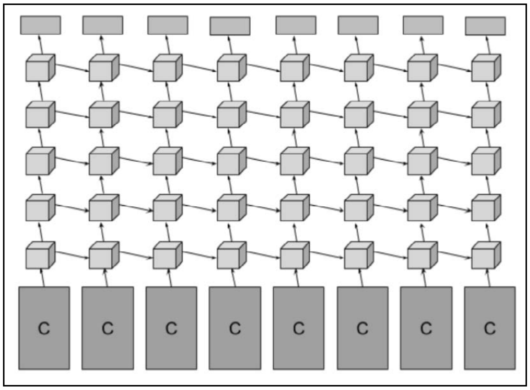
添加用于特征池化的 LSTM 可以提供更好的性能。这些特征以各种方式池化，如图 9-6 所示。



* 经 Ng 等人许可转载

图　9-6

如图 9-6 所示，卷积特征可以以几种不同的方式进行聚合。池化在全连接层之后完成。该方法在 Sports1M 数据集和 UCF101 数据集中分别达到了 73.1%和 88.6%的准确率。LSTM 方法如图 9-7 所示。



* 经 Ng 等人许可转载

图 9-7

因为使用了几个 LSTM，这个模型的计算复杂度结果很高。

3. 双 CNN 流动作识别

视频中目标的运动具有关于动作非常好的信息。目标的运动可以通过光流来量化。Simonyan 和 Zisserman 提出了一种动作识别方法，该方法使用来自图像的两个流和光流。

光流通过量化观察者和场景之间的相对运动来测量动作。通过运行以下命令可以获得光流：

```
p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None,
**lk_params)
```

一个流接收一个单独的帧并使用常规 CNN 预测动作。另一个流需要多个帧并计算光流。光流通过 CNN 进行预测。这两个预测都显示在图 9-8 中。

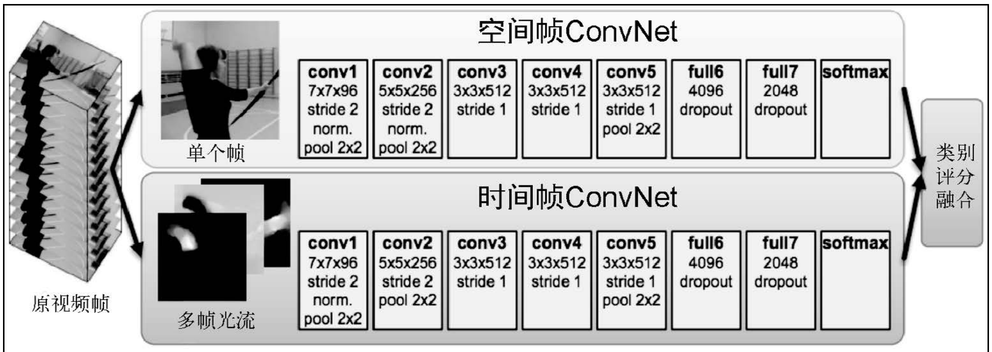


Figure 1: 视频分类的双流架构

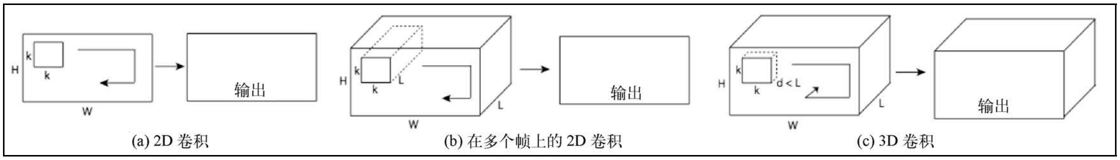
* 经 Simonyan 和 Zisserman 许可转载

图 9-8

这两种预测都可以与最终预测相结合。

4. 使用 3D 卷积进行时间学习

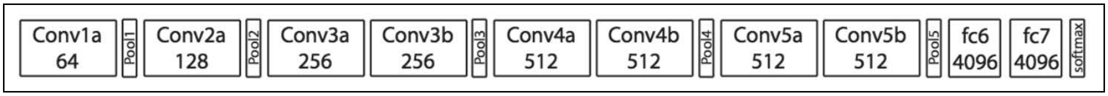
视频可以使用 3D 卷积进行分类。3D 卷积操作将容积作为输入并输出容积，而 2D 卷积可以接收一个 2D 或容积输出并输出一个 2D 图像。差异如图 9-9 所示。



* 经 Tran 等人许可转载

图　9-9

图 a 和图 b 属于 2D 卷积，输出总是一幅图像。同时，3D 卷积输出容积。不同之处是使用核函数在 3 个方向上的卷积操作。Tran 等人使用 3D 卷积进行视频分类。3D 卷积模型如图 9-10 所示。



* 经 Tran 等人许可转载

图　9-10

以下是使用 3D 卷积的模型的代码片段：

```
net = tf.keras.models.Sequential()
net.add(tf.keras.layers.Conv3D(32,
                               kernel_size=(3, 3, 3),
                               input_shape=(input_shape)))
net.add(tf.keras.layers.Activation('relu'))
net.add(tf.keras.layers.Conv3D(32, (3, 3, 3)))
net.add(tf.keras.layers.Activation('softmax'))
net.add(tf.keras.layers.MaxPooling3D())
net.add(tf.keras.layers.Dropout(0.25))

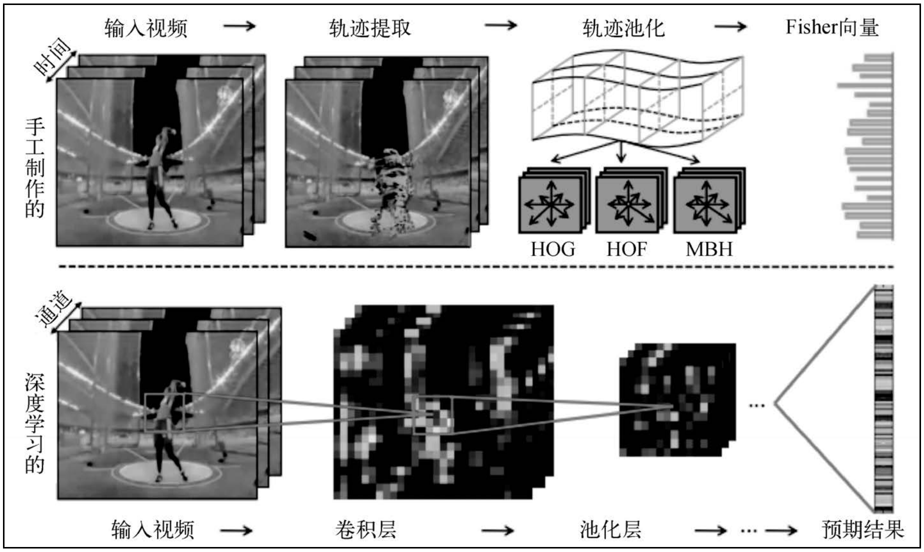
net.add(tf.keras.layers.Conv3D(64, (3, 3, 3)))
net.add(tf.keras.layers.Activation('relu'))
net.add(tf.keras.layers.Conv3D(64, (3, 3, 3)))
net.add(tf.keras.layers.Activation('softmax'))
net.add(tf.keras.layers.MaxPool3D())
net.add(tf.keras.layers.Dropout(0.25))

net.add(tf.keras.layers.Flatten())
net.add(tf.keras.layers.Dense(512, activation='sigmoid'))
net.add(tf.keras.layers.Dropout(0.5))
net.add(tf.keras.layers.Dense(no_classes, activation='softmax'))
net.compile(loss=tf.keras.losses.categorical_crossentropy,
            optimizer=tf.keras.optimizers.Adam(), metrics=['accuracy'])
```

3D 卷积需要大量的计算能力。3D 卷积在 Sports1M 数据集上达到了 90.2%的准确率。

5. 使用轨迹进行分类

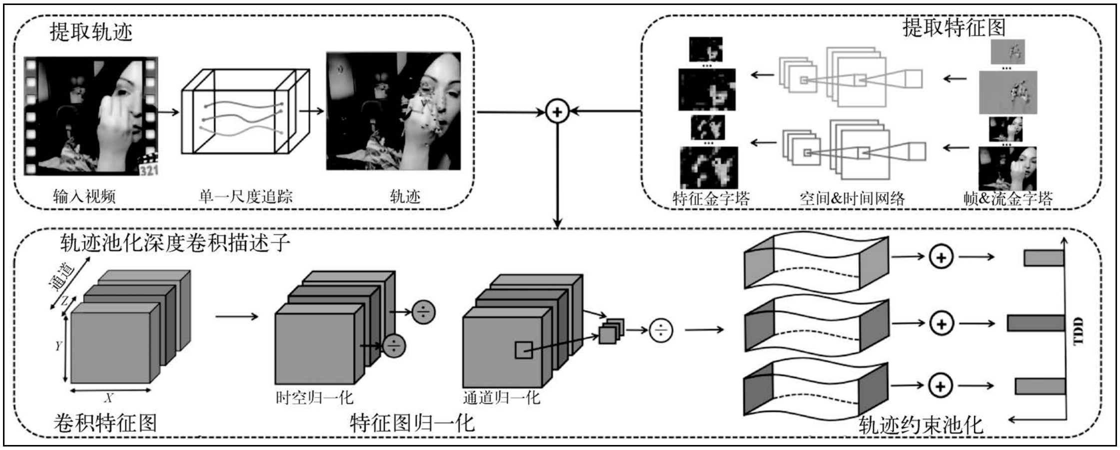
Wang 等人使用身体部分的轨迹（trajectory）来分类动作。这项工作结合了手工制作和深度学习的特征进行最终的预测。图 9-11 显示了分类。



* 经 Wang 等人许可转载

图 9-11

手工制作的特征是 Fisher 向量，特征来自 CNN。图 9-12 演示了轨迹和特征图的提取。



* 经 Wang 等人许可转载

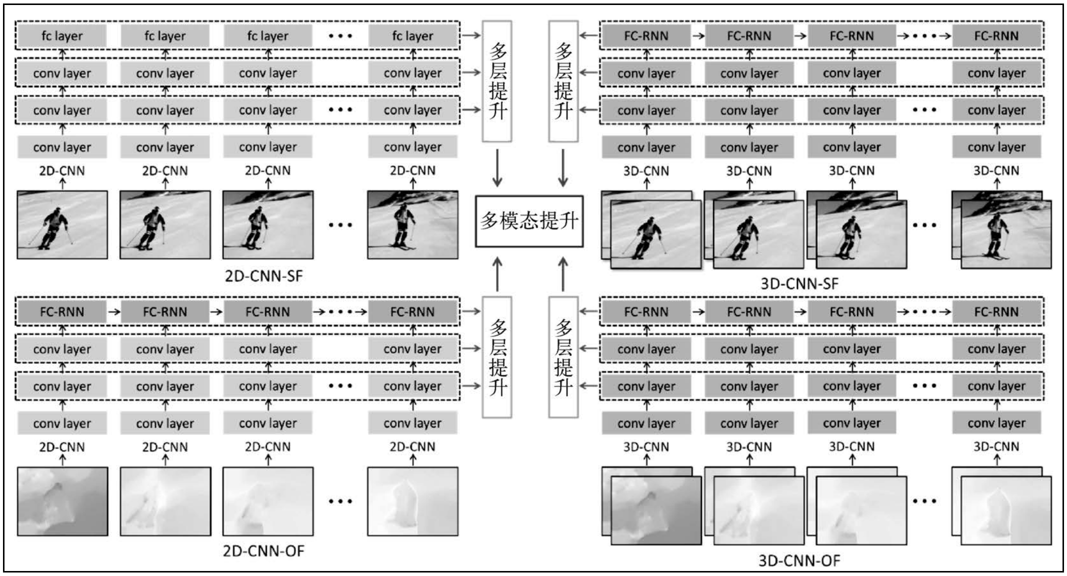
图 9-12

轨迹和特征图在时间上组合以形成对时间片段的最终预测。

6. 多模态融合

Yang 等人提出了一种多模态融合，有 4 种模型用于视频分类。这 4 种模型是三维卷积特征、二维光流、三维光流和二维卷积特征。

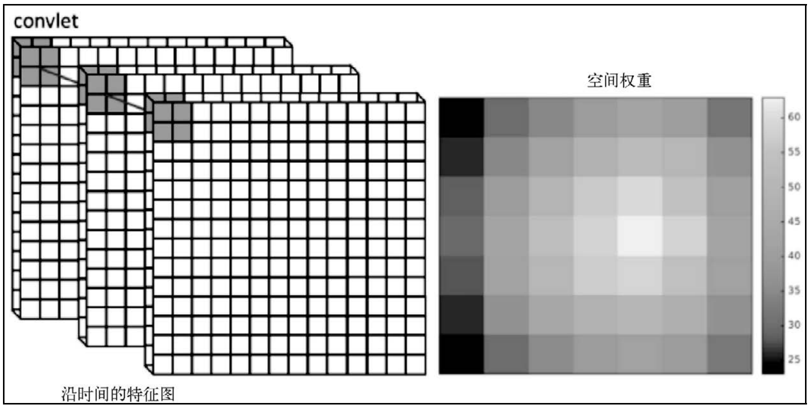
该方法中的数据流如图 9-13 所示。



* 经 Yang 等人许可转载

图　9-13

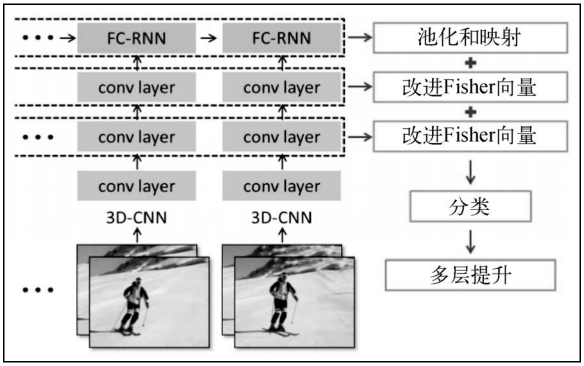
现在，我们来了解一下 Convlet。Convlet 是来自单个核的小的卷积输出。图 9-14 显示了使用 Convlet 在卷积层中学习空间权重的方法。



* 经 Yang 等人许可转载

图　9-14

空间权重指示卷积层中局部空间区域的区别性或重要性如何。图 9-15 是融合多层表征的图示，在卷积层和全连接层的多个层完成。



* 经 Yang 等人许可转载

图 9-15

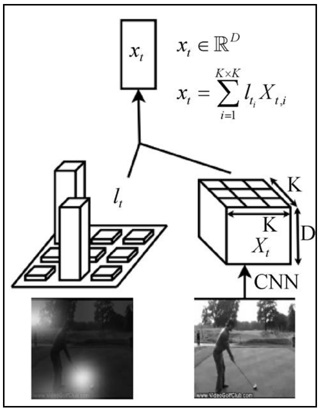
提升（boosting）机制用于组合预测，可以把多个模型预测合并为最终预测。

7. 注意区域分类

注意力机制可以用于分类。注意力机制会复制人在关注某些区域进行识别活动时的行为。该机制对某些区域赋予了更高的权重。赋予权重的方法是从数据中训练学习到的。注意力机制主要有以下两种。

- 软注意力：本质上具有确定性，因此可以通过反向传播来学习。
- 硬注意力：本质上具有随机性，需要复杂的机制来学习。由于需要采样数据，它还很昂贵。

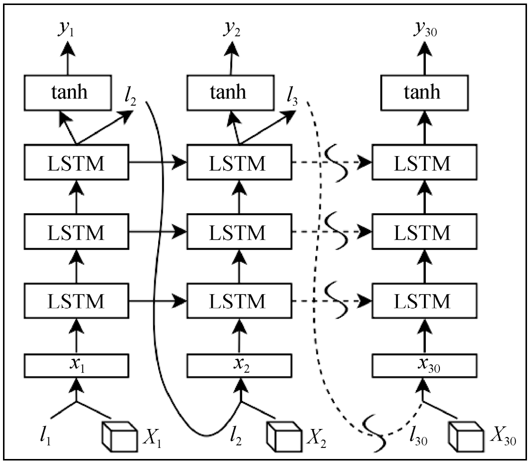
图 9-16 是一个软注意力的可视化。



* 经 Sharma 等人许可转载

图 9-16

CNN 特征根据注意力进行计算和加权。CNN 特征给予某些区域的注意力或权重可用于可视化。Sharma 等人使用这个想法来分类视频。LSTM 被用来获取卷积特征。LSTM 通过注意后续帧来预测区域，如图 9-17 所示。



* 经 Sharma 等人许可转载

图　9-17

LSTM 堆叠预测位置和标签，每个堆叠都有三个 LSTM。LSTM 堆叠的输入是一个卷积特征方块和位置。位置可能性是注意力权重。注意力的使用提高了准确性，也可作为一种预测的可视化方法。

我们已经了解了各种视频分类方法，接下来将介绍视频中的其他应用。

9.2　将基于图像的方法扩展到视频

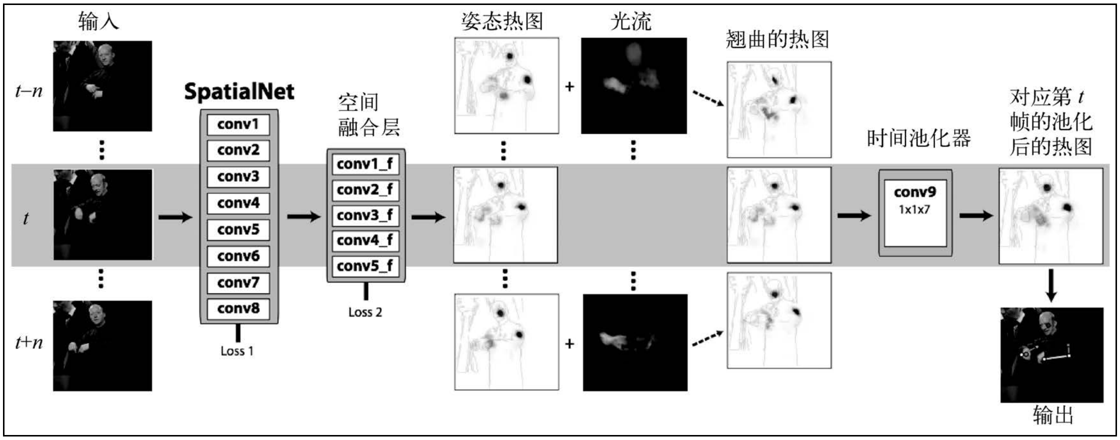
图像可用于姿态估计、风格迁移、图像生成、分割、题注等。同样，这些应用在视频中也有一席之地。使用时间信息可以改善来自图像的预测，反之亦然。本节将介绍如何将这些应用扩展到视频。

9.2.1　人体姿态回归

人体姿态估计是视频数据的重要应用，并且可以改进其他任务，如动作识别。首先，我们来看看可用于姿态估计的数据集的描述。

- Poses in the wild dataset：包含用人体姿态标注的 30 个视频。数据集用人体上肢关节标注。
- Frames Labeled In Cinema（FLIC）：从 30 部电影中获得的人体姿态数据集。

Pfister 等人提出了一种预测视频中人体姿态的方法。图 9-18 是人体姿态回归的流水线。



* 经 Pfister 等人许可转载

图 9-18

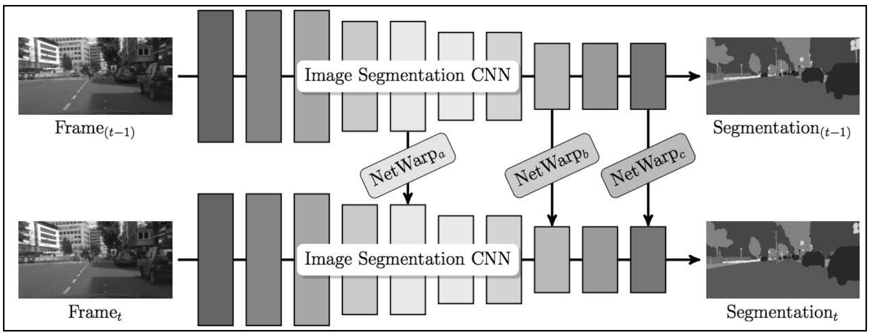
来自视频的帧被接收并通过卷积网络。各层融合，获得姿态热图。姿态热图与光流组合以获得翘曲的热图。在一定的时间段内翘曲的热图经过池化产生池化的热图，从而得到最终的姿态。

面部特征点跟踪

视频中的人脸分析需要人脸检测、特征点检测、姿态估计、验证等。计算特征点对于捕捉面部动画、人机交互和人类活动识别尤为重要。这可以通过视频来计算，而不是通过帧计算。Gu 等人提出了一种 RNN 方法，在视频中使用面部特征点检测和跟踪的联合估计。其结果优于帧级别的预测和其他以前的模型。特征点由 CNN 计算，并且时间方面被编码在 RNN 中。训练使用合成数据。

9.2.2 视频分割

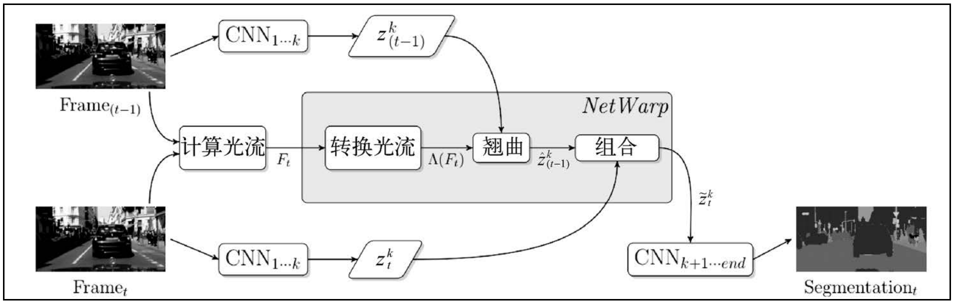
使用时间信息时，可以更好地分割视频。Gadde 等人提出了一种通过翘曲来组合时间信息的方法。该解决方案分割两个框架并组合翘曲，如图 9-19 所示。



* 经 Gadde 等人许可转载

图 9-19

图 9-20 显示了翘曲网络。



* 经 Gadde 等人许可转载

图　9-20

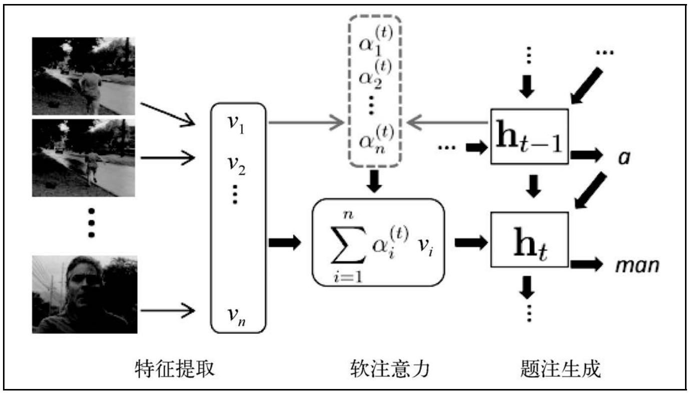
光流在两帧之间计算，与翘曲相结合。翘曲模块获取光流，对其进行变换并将其与翘曲表征组合。

9.2.3　视频题注

第 7 章说明了文字和图像组合的几种方法。类似地，可以为视频生成题注，描述上下文。让我们来看看可用于视频题注的数据集列表。

- Microsoft Research-Video To Text（MSR-VTT）拥有 200 000 个视频剪辑和句子对。
- MPII Movie Description Corpus（MPII-MD）有 68 000 个句子和 94 部电影。
- Montreal Video Annotation Dataset（M-VAD）有 49 000 个剪辑。
- YouTube2Text 有 1970 个视频和 80 000 个描述。

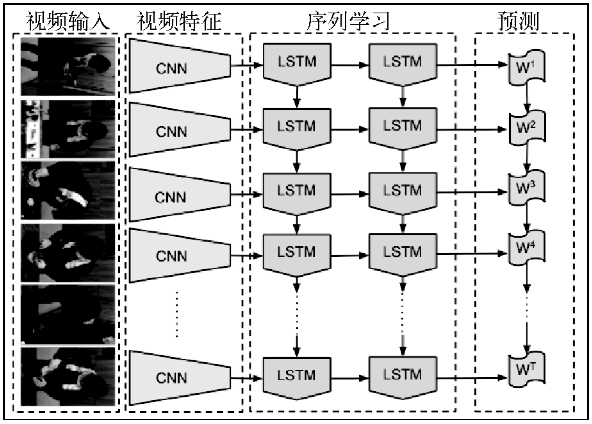
Yao 等人提出了一种视频题注的方法。一个为动作识别的 3D 卷积网络被训练用于提取局部时间特征。然后使用注意力机制在特征上用一个 RNN 生成文本。该过程如图 9-21 所示。



* 经 Yao 等人许可转载

图　9-21

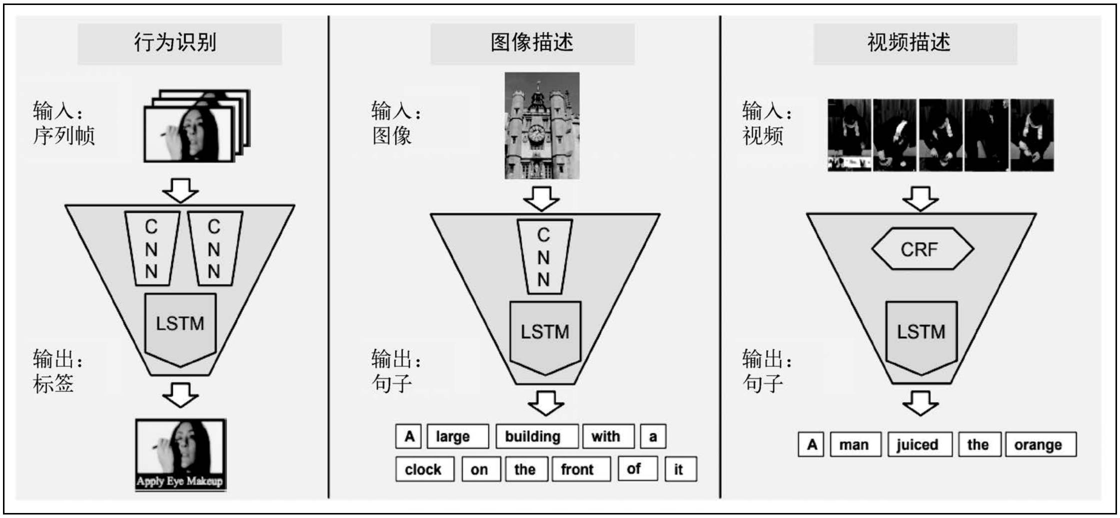
Donahue 等人提出了另一种用于视频题注（或描述）的方法，该方法使用 LSTM 和卷积特征。这与前面的方法类似，只是我们在这里使用了二维卷积特征，如图 9-22 所示。



* 经 Donahue 等人许可转载

图　9-22

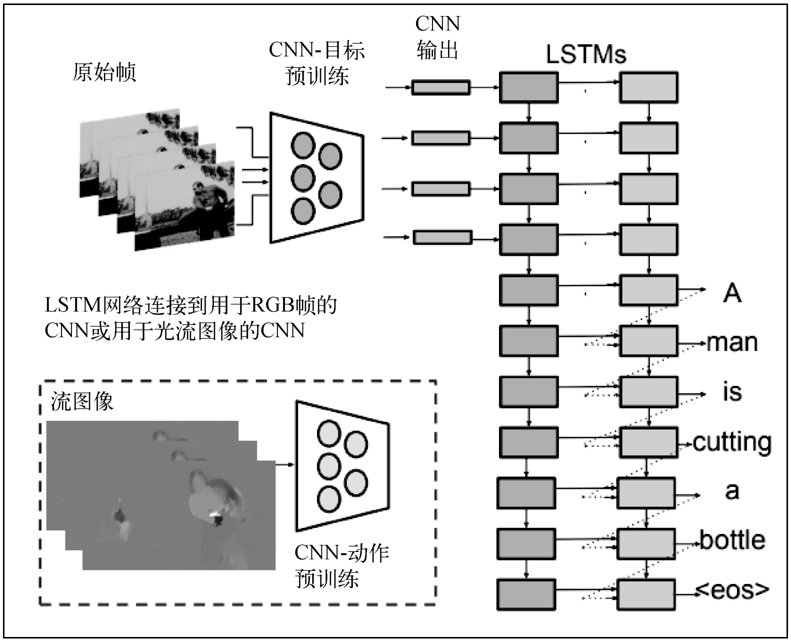
我们有几种方法可以将文本与图像相结合，例如行为识别、图像描述和视频描述技术。图 9-23 展示了这些技术。



* 经 Donahue 等人许可转载

图　9-23

Venugopalan 等人提出了一种使用编码器–解码器方法进行视频题注的方法。图 9-24 是该技术的可视化。



* 经 Venugopalan 等人许可转载

图　9-24

TIP 对于这个方法，CNN 可以在图像的帧或图像的光流上进行计算。

9.2.4　视频生成

视频可以使用生成模型以无监督的方式生成，未来帧可以使用当前帧进行预测。Ranzato 等人提出了一种由语言模型启发的视频生成方法。利用一个 RNN 模型接收图像的一个图块并预测下一个图块。

9.3　小结

本章讨论了与视频分类有关的各种主题。我们了解了如何将视频分割为多个帧，并使用图像中的深度学习模型来完成各种任务；学习了一些特定于视频的算法，如跟踪目标；掌握了如何将基于视频的解决方案应用于各种场景，如动作识别、手势识别、安全应用程序和入侵检测。

在下一章中，我们将学习如何将上一章中训练好的模型部署到各种云平台和移动平台上的生产环境中。我们将了解不同硬件如何影响延迟和吞吐量方面的性能。

第 10 章 部 署

在本章中，我们将学习如何在各种平台上部署经过训练的模型，以获得最大的吞吐量和最小的延迟；了解各种硬件（如 GPU 和 CPU）的性能；遵循在 AWS、Google 云平台以及 Android、iOS 和 Tegra 等移动平台上部署 TensorFlow 的步骤。

本章涉及以下主题：

- 了解影响深度学习模型训练和推断性能的因素；
- 通过各种方法提高性能；
- 查看各种硬件的基准并学习如何调整它们以获得最佳性能；
- 使用各种云平台进行部署；
- 使用各种移动平台进行部署。

10.1 模型的性能

性能对于深度学习模型的训练和部署都很重要。由于大数据或大型模型架构，训练通常需要更多时间。由此产生的模型可能规模更大，因此在存在 RAM 限制的移动设备中使用时会有问题。更长的计算时间导致更高的基础设施成本。推断时间在视频应用中至关重要。由于前面提到的性能重要性，在本节中，我们将研究提高性能的技术。降低模型复杂度是一个简单的选择，但会导致准确率下降。本节，我们将重点关注既不会让准确率大幅下降又能提高性能的方法。下一节将讨论量化选项。

10.1.1 量化模型

深度学习模型的权重有 32 位浮点值。当权重被量化为 8 位时，准确率略有下降，因此在部署时不会被注意到。权重精确度似乎对深度学习模型的准确率性能影响较小。这个关于深度学习的想法很有趣，它在模型规模变得至关重要时会很有用。通过用 8 位值替换 32 位浮点值，可以显著减小模型规模，并且加快推断速度。实现模型量化时有很多选择。权重可以以 8 位存储，但推断操作可以以 32 位浮点值执行。对于量化大小，架构的每个组件可能会有不同的表现，因此，

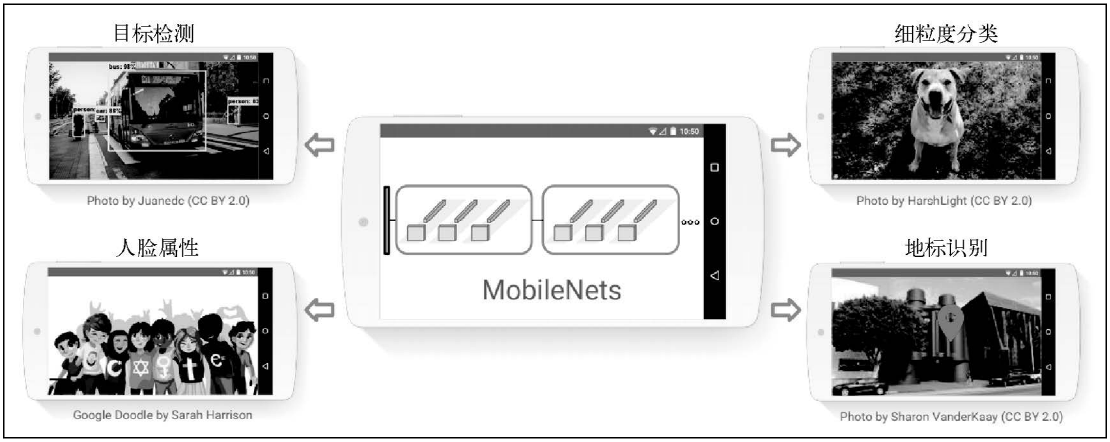
根据层的不同，可以选择 32 位、16 位或 8 位值。

量化能起效果的原因有很多。通常，深度学习模型经过训练可以抑制图像中的噪声，因此可以认为是稳健的。推断计算可能含有冗余信息，而冗余信息可能由于量化而被移除。

最新的 CPU 和 RAM 硬件都被调整为浮点运算，因此在这种硬件中的量化效果可能不太明显。随着越来越多的硬件为此而引入，这种情况正在发生变化。在 GPU 中，可以注意到内存和速度的显著差异，因为它们现在适用于较低精确度的浮点运算。还有其他特殊的硬件可用于运行低精确度浮点运算。

10.1.2　MobileNets

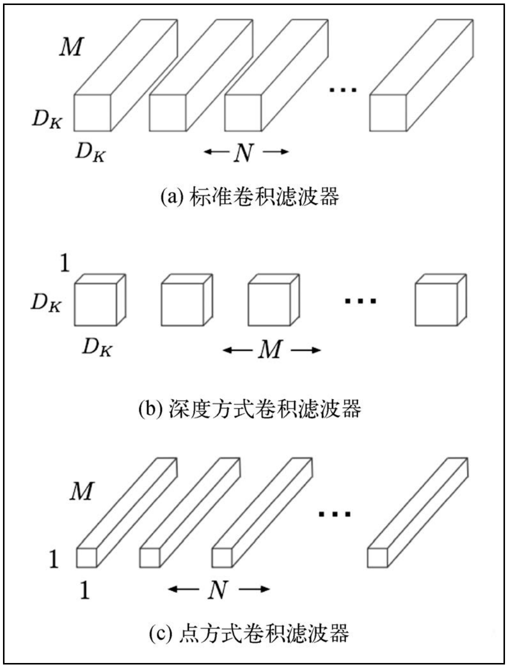
Howard 和其他人推出了一类名为 MobileNets 的新模型，可用于移动和嵌入式应用。MobileNets 可用于不同的应用，如目标检测、地标识别、人脸属性、细粒度分类，如图 10-1 所示。



* 经 Howard 等人许可转载

图　10-1

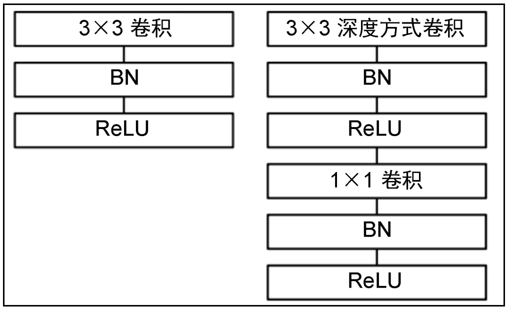
MobileNets 通过用深度方式卷积滤波器和点方式卷积滤波器替换标准卷积滤波器来减小模型的规模和减少计算量，如图 10-2 所示。



* 经 Howard 等人许可转载

图　10-2

批归一化和激活层被添加到深度方式卷积滤波器和点方式卷积滤波器中，如图 10-3 所示。

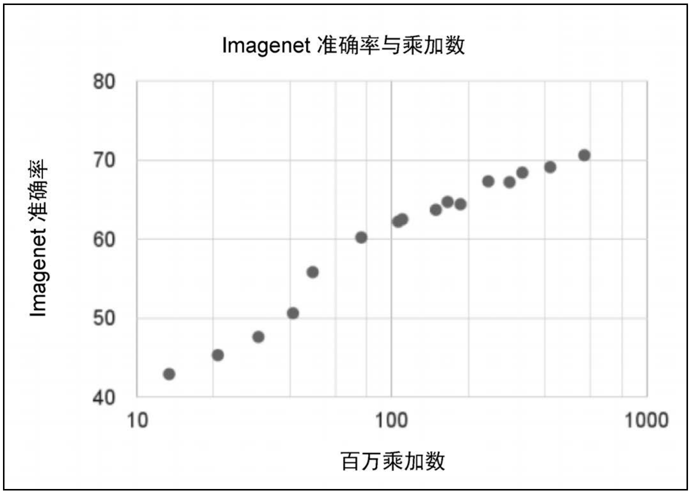


* 经 Howard 等人许可转载

图　10-3

有两个参数影响模型的选择。

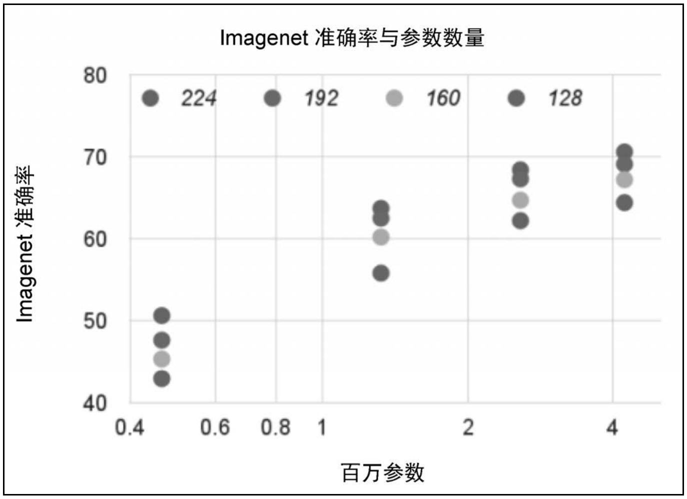
- **乘法和加法的数量**：准确率和加乘数量之间的权衡如图 10-4 所示。



* 经 Howard 等人许可转载

图　10-4

- **模型中的参数数量**：权衡如图 10-5 所示。



* 经 Howard 等人许可转载

图　10-5

MobileNets 已经表明，可以通过小幅降低移动和嵌入式设备上的准确率来减小模型计算和规模。可以在 Howard 等人的文章中看到模型和准确率之间的确切权衡。

10.2　云部署

对于某些应用，模型必须在云上部署。接下来将介绍主要的云服务提供商。

10.2.1　AWS

亚马逊 Web 服务（AWS）扩展了对基于 TensorFlow 模型的开发和部署的支持。在 https://aws.amazon.com/注册 AWS 并选择其中一个 **Amazon Machine Images**（AMI）。AMI 是安装了所有必需软件的机器的映像。你不必担心安装软件包。AWS 提供深度学习 AMI（DLAMI），以便于训练和部署深度学习模型。有几种选择。在这里，我们将使用 Conda，因为它带有运行 TensorFlow 所需的几个软件包。Python 有两个选项：版本 2 和版本 3。使用以下代码在 CUDA 8 的 Python 3 上用 Keras 2 激活 TensorFlow。

```
source activate tensorflow_p36
```

使用以下代码在 CUDA 8 的 Python 2 上用 Keras 2 激活 TensorFlow。

```
source activate tensorflow_p27
```

TIP　可以访问 https://aws.amazon.com/tensorflow/了解更多详情和教程。

还可以按照以下步骤启动一个**虚拟机**（VM）。

(1) 转到 aws.amazon.com 并使用你的亚马逊账户登录。
(2) 从登录页面选择 **Launch a virtual machine**（启动虚拟机），如图 10-6 所示。

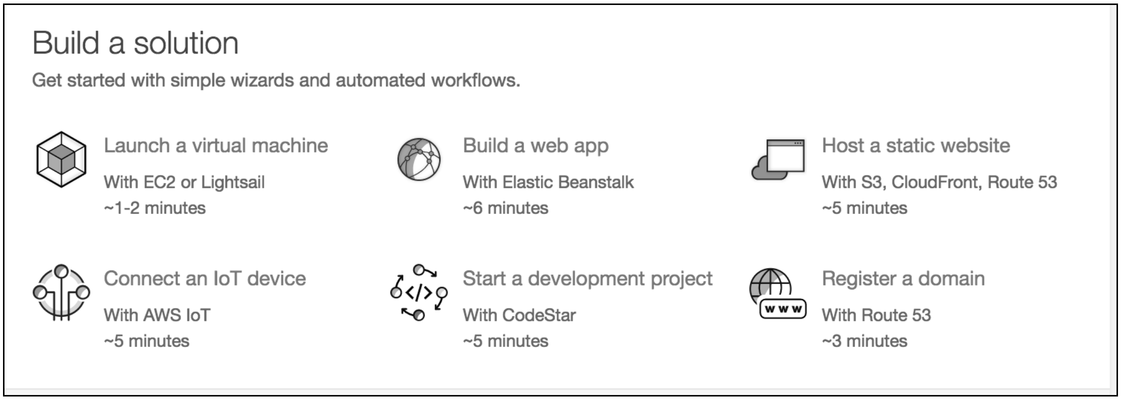


图　10-6

(3) 在下一个窗口中，通过单击如图 10-7 所示的 **Get Started**（开始）选择 EC2 Instance（EC2 实例）。

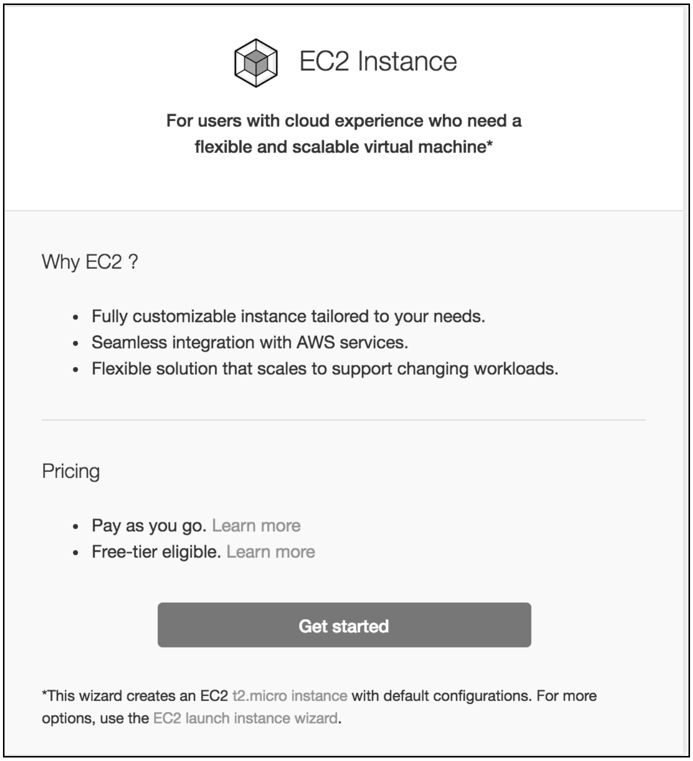


图 10-7

(4) 为 EC2 实例命名，如图 10-8 所示。



图 10-8

(5) 选择操作系统的类型，如图 10-9 所示。

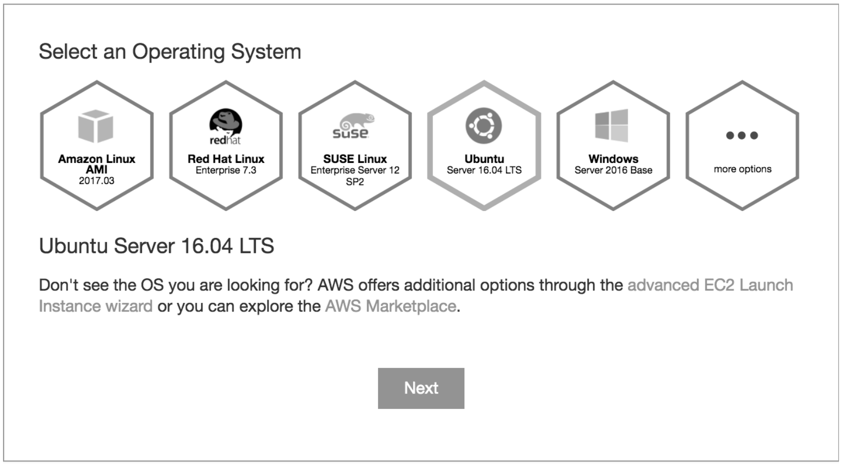


图 10-9

(6) 选择实例类型。实例类型表示具有不同 RAM 和 CPU 大小的配置类型。GPU 也有几个选项可供选择。选择实例类型并单击“Next”按钮，如图 10-10 所示。

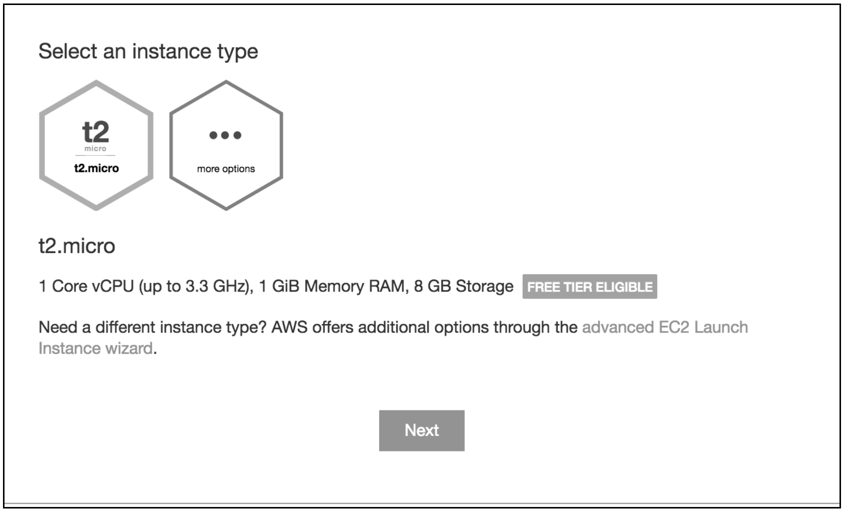


图 10-10

(7) 创建用于登录的保密增强邮件（PEM）安全证书文件，如图 10-11 所示。

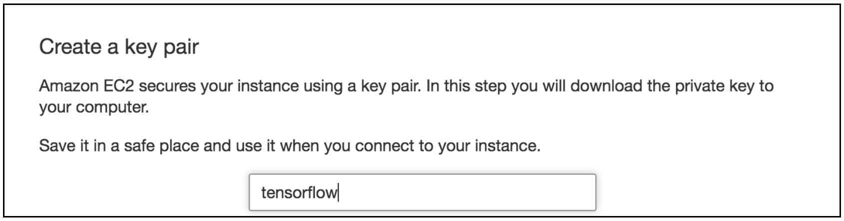


图　10-11

(8) 创建实例需要一些时间，最后会显示完成的状态，如图 10-12 所示。

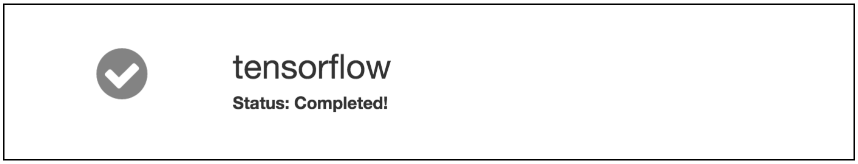


图　10-12

(9) 接下来，单击“Proceed to EC2 console”按钮，如图 10-13 所示。

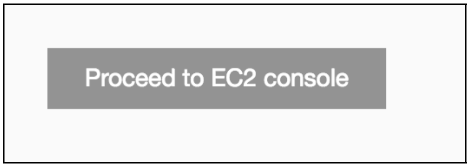


图　10-13

(10) 现在将创建实例，单击“Connect”按钮，如图 10-14 所示。

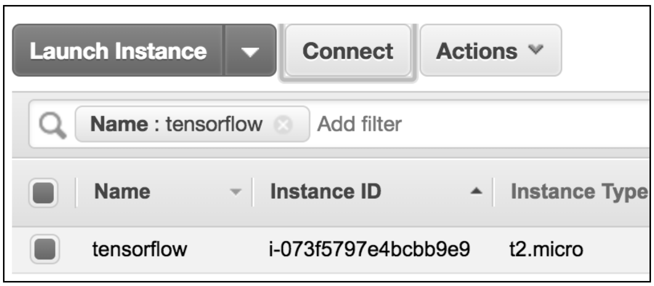


图　10-14

(11) 接下来，该实例必须连接到虚拟机的命令提示符。连接所需的说明在此给出。你需要在前面的步骤中下载“pem”文件。按照显示的说明连接到系统，如图 10-15 所示。

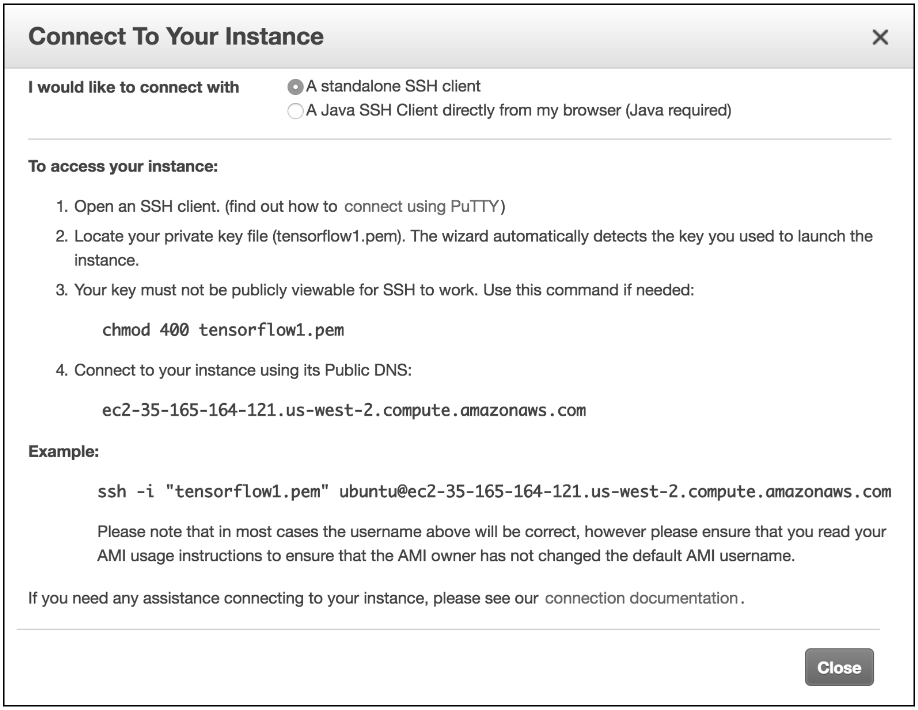


图 10-15

(12) 完成后，单击“Actions|Instance State|Terminate”终止实例，如图 10-16 所示。

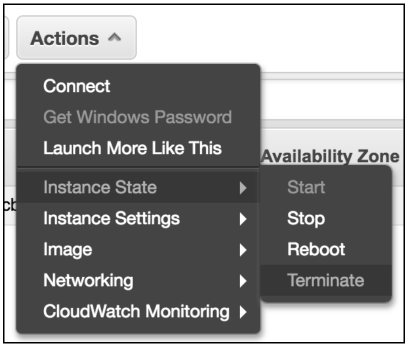


图 10-16

安装和执行步骤可以根据第 1 章进行。

10.2.2 Google 云平台

Google Cloud Platform（GCP）是 Google 提供的云平台，具有与 AWS 相似的功能。通过执

行以下步骤，可以使用简单的虚拟机来训练模型。

(1) 使用 cloud.google.com 访问 Google Cloud Platform，并使用你的 Gmail 账户登录平台。

(2) 现在点击“GO TO CONSOLE”按钮进入控制台，如图 10-17 所示。



图　10-17

(3) 进入控制台后，单击右上角菜单中的“Compute Engine|VM instance”转至 VM 创建页面，如图 10-18 所示。

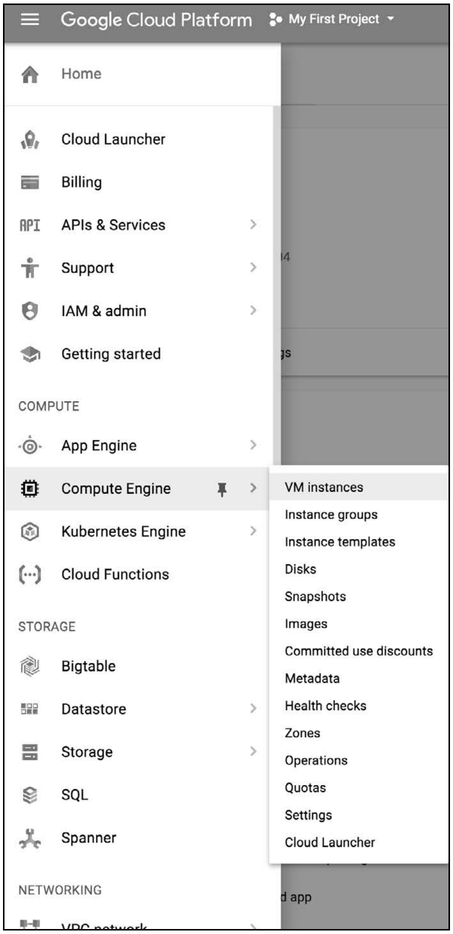


图　10-18

(4) 然后单击“CREATE INSTANCE”按钮，以创建所需的实例，如图 10-19 所示。



图　10-19

(5) 接下来，可以使用配置选择实例类型。区域参数通知地区实例将被部署。通过选择靠近用户的区域，可以节省延迟时间。机器类型可以根据所需的 RAM 和 CPU 进行定制。GPU 也可以选择更快的训练。选择实例的大小，然后单击“Create”按钮，如图 10-20 所示。

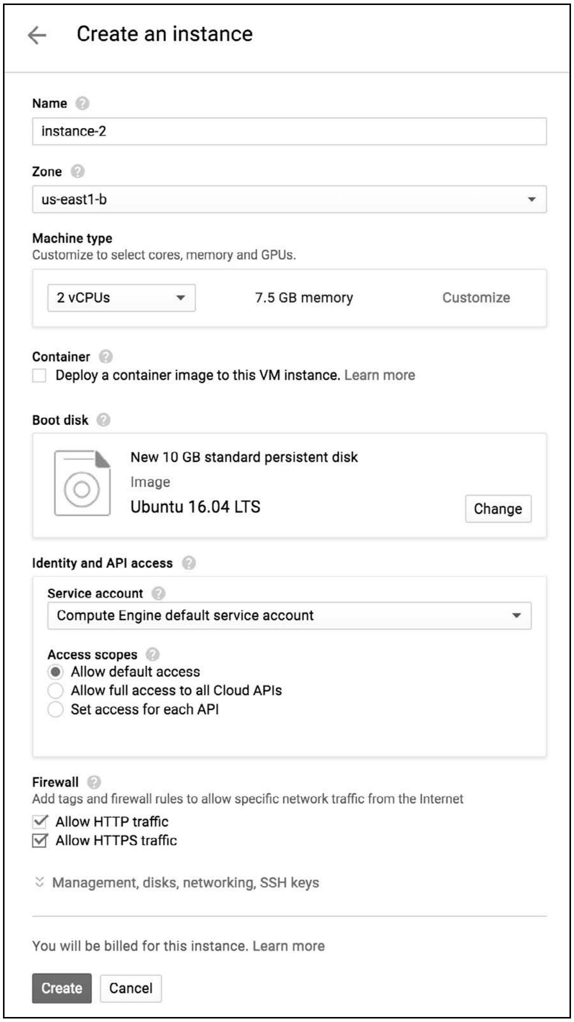


图　10-20

(6)创建实例需要几分钟的时间。然后点击实例的 SSH 下拉列表并选择“Open in browser window”选项，如图 10-21 所示在浏览器中打开控制台。

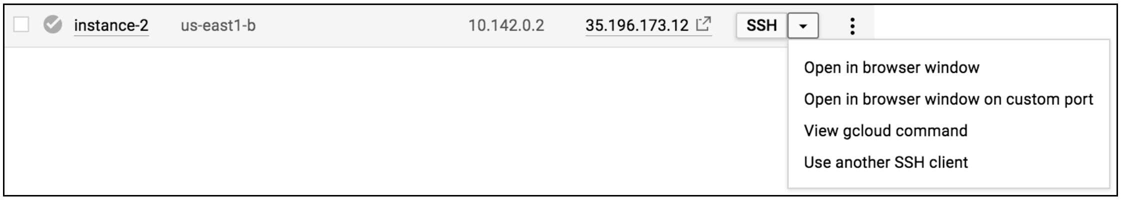


图　10-21

使用 shell，你可以安装 TensorFlow 并训练或部署模型。有很多选项可以从虚拟机的配置中进行选择。根据成本和时间的权衡，可以选择相应的配置。

GCP 有一台云计算机学习引擎，可以帮助我们使用 TensorFlow。GCP 的三个组件可以一起用于构建训练和部署基础架构。

(1) 用于预处理图像的 Cloud DataFlow。
(2) 用于训练和部署模型的 Cloud Machine Learning Engine。
(3) 用于存储训练数据、代码和结果的 Google Cloud Storage。

TIP　可以在 https://cloud.google.com/ml-engine/docs/flowers-tutorial 上找到使用 Cloud Machine Learning Engine 构建自定义图像分类模型的优秀教程。

10.3　在设备中部署模型

TensorFlow 模型也可以部署在移动设备中。移动设备包括智能手机、无人机、家庭机器人等。数十亿智能手机可以使用基于深度学习的计算机视觉应用程序。人们可以拍摄照片并进行搜索，播放带有场景标记的视频，等等。在移动设备中部署意味着深度学习模型出现在设备上，并在设备上进行推断。设备上部署的模型有助于解决隐私问题。在接下来的主题中，我们将讨论如何在各种移动平台上部署它们。

10.3.1　Jetson TX2

Jetson TX2 是由 NVIDIA 公司提供的高效的 AI 计算嵌入设备。Jetson TX2 轻量级、简洁，适用于无人机、公共场所等的部署。它还预装了 TensorFlow 的 TensorRT 运行系统。你可以购买 Jetson，并在安装 TensorFlow 之前安装 Ubuntu、CUDA 和 CUDNN。克隆 https://github.com/jetsonhacks/installTensorFlowTX2 并在命令提示符处输入以下命令。

(1) 首先，安装先决条件。

```
./installPrerequisites.sh
```

(2) 然后，克隆 TensorFlow。

```
./cloneTensorFlow.sh
```

(3) 接下来，设置所需的环境变量。

```
./setTensorFlowEV.sh
```

(4) 使用以下代码构建 TensorFlow。

```
./buildTensorFlow.sh
```

(5) 接下来，将程序包处理成 wheel 文件。

```
./packageTensorFlow.sh
```

(6) 最后，安装 TensorFlow。

```
pip install $HOME/tensorflow-1.0.1-cp27-cp27mu-linux_aarch64.whl
```

借助这些步骤，我们就可以在 Jetson TX2 中安装 TensorFlow 了。

10.3.2 Android

任何 Android 应用程序都可以使用 TensorFlow，并且可以在 tensorflow.org 网站中找到要构建的细节。官方示例可以在 https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/android 中找到。如果你具有 Android 编程经验，在 Android 设备中实现 Tensorflow 的步骤如下。

(1) 使用第 3 章中介绍的步骤，将 TensorFlow 模型导出为.pb 文件。
(2) 生成.so 和.jar 文件，这是二进制文件。
(3) 编辑 gradle 文件以启用加载库。
(4) 加载并运行 Android 应用程序文件。

10.3.3 iPhone

Apple 使用 CoreML 框架将机器学习整合到 iPhone 的应用程序中。Apple 提供了可直接整合到应用程序中的标准模型列表。我们可以使用 TensorFlow 训练自定义的深度学习模型，并在 iPhone 中使用它。为了部署自定义模型，必须在 CoreML 框架模型中隐藏 TensorFlow。Google 发布了 https://github.com/tf-coreml/tf-coreml，用于将 TensorFlow 模型转换为 CoreML 模型。TFcoreML 可以使用以下代码进行安装：

```
pip install -U tfcoreml
```

该模型可以使用以下代码导出：

```
import tfcoreml as tf_converter
tf_converter.convert(tf_model_path='tf_model_path.pb',
                     mlmodel_path='mlmodel_path.mlmodel',
                     output_feature_names=['softmax:0'],
                     input_name_shape_dict={'input:0': [1, 227, 227, 3]})
```

iPhone 可利用输出的模型进行预测。

10.4　小结

在本章中，我们学习了如何在各种平台和设备上部署经过训练的深度学习模型，以及针对这些平台获得最佳性能的步骤和指南，还了解了 MobileNets 的优点，即通过略微牺牲准确率来缩短推断时间。

深度学习已在人工智能的多个应用领域（尤其是计算机视觉）中展示了其强大功能。计算机视觉是理解和处理图像的科学，并且在机器人技术、自动化等领域有广泛的应用。

本书通过实例展示了如何利用深度学习的力量来开发计算机视觉应用程序，介绍了与图像分类、图像检索、目标检测、语义分割、图像题注、生成模型等有关的各种技术。你将使用流行的 Python 库（例如 TensorFlow 和 Keras）探索计算机视觉应用程序，从而掌握各种深度学习算法及其实现。

- 使用TensorFlow和Keras设置深度学习的环境
- 探索图像分类方法并训练一个深度学习模型
- 使用预训练的卷积神经网络模型进行图像检索
- 了解检测方法并实施行人检测
- 了解图像题注中的各种问题及其实现
- 训练一个使用生成对抗网络生成图像的模型
- 了解视频分类的方法及其实现
- 部署深度学习模型并进行优化

Packt
www.packtpub.com

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议　计算机 / 机器学习 / 深度学习

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-53158-2

ISBN 978-7-115-53158-2
定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks